

---

# Koalas

*Release 1.8.2*

**The Koalas Team**

**Sep 21, 2023**



# CONTENTS

<b>1</b>	<b>Getting started</b>	<b>3</b>
1.1	Installation	3
1.1.1	Python version support	3
1.1.2	Installing Koalas	3
1.1.3	Installing PySpark	4
1.1.4	Dependencies	5
1.2	10 minutes to Koalas	5
1.2.1	Object Creation	6
1.2.2	Viewing Data	8
1.2.3	Missing Data	9
1.2.4	Operations	10
1.2.5	Grouping	11
1.2.6	Plotting	12
1.2.7	Getting data in/out	12
1.3	Koalas Talks and Blogs	13
1.3.1	Blog Posts	13
1.3.2	Data + AI Summit 2020 EUROPE (Nov 18-19, 2020)	14
1.3.3	Spark + AI Summit 2020 (Jun 24, 2020)	14
1.3.4	Webinar @ Databricks (Mar 27, 2020)	14
1.3.5	PyData New York 2019 (Nov 4, 2019)	14
1.3.6	Spark + AI Summit Europe 2019 (Oct 16, 2019)	14
1.3.7	PyBay 2019 (Aug 17, 2019)	14
1.3.8	Spark + AI Summit 2019 (Apr 24, 2019)	14
<b>2</b>	<b>User Guide</b>	<b>15</b>
2.1	Options and settings	15
2.1.1	Getting and setting options	15
2.1.2	Operations on different DataFrames	16
2.1.3	Default Index type	17
2.1.4	Available options	19
2.2	Working with pandas and PySpark	19
2.2.1	pandas	19
2.2.2	PySpark	20
2.3	Transform and apply a function	21
2.3.1	<code>transform</code> and <code>apply</code>	21
2.3.2	<code>koalas.transform_batch</code> and <code>koalas.apply_batch</code>	23
2.4	Type Support In Koalas	25
2.4.1	Type casting between PySpark and Koalas	25
2.4.2	Type casting between pandas and Koalas	27
2.4.3	Internal type mapping	28

2.5	Type Hints In Koalas . . . . .	29
2.5.1	Koalas DataFrame and Pandas DataFrame . . . . .	30
2.5.2	Type Hinting with Names . . . . .	30
2.6	From/to other DBMSes . . . . .	31
2.6.1	Reading and writing DataFrames . . . . .	32
2.7	Best Practices . . . . .	33
2.7.1	Leverage PySpark APIs . . . . .	33
2.7.2	Check execution plans . . . . .	34
2.7.3	Use checkpoint . . . . .	34
2.7.4	Avoid shuffling . . . . .	35
2.7.5	Avoid computation on single partition . . . . .	35
2.7.6	Avoid reserved column names . . . . .	35
2.7.7	Do not use duplicated column names . . . . .	36
2.7.8	Specify the index column in conversion from Spark DataFrame to Koalas DataFrame . . . . .	36
2.7.9	Use distributed or distributed-sequence default index . . . . .	36
2.7.10	Reduce the operations on different DataFrame/Series . . . . .	37
2.7.11	Use Koalas APIs directly whenever possible . . . . .	37
2.8	FAQ . . . . .	38
2.8.1	What's the project's status? . . . . .	38
2.8.2	Is it Koalas or koalas? . . . . .	38
2.8.3	Should I use PySpark's DataFrame API or Koalas? . . . . .	38
2.8.4	Does Koalas support Structured Streaming? . . . . .	39
2.8.5	How can I request support for a method? . . . . .	39
2.8.6	How is Koalas different from Dask? . . . . .	39
2.8.7	How can I contribute to Koalas? . . . . .	39
2.8.8	Why a new project (instead of putting this in Apache Spark itself)? . . . . .	39
<b>3</b>	<b>API Reference</b> . . . . .	<b>41</b>
3.1	Input/Output . . . . .	41
3.1.1	Data Generator . . . . .	41
3.1.2	Spark Metastore Table . . . . .	42
3.1.3	Delta Lake . . . . .	44
3.1.4	Parquet . . . . .	46
3.1.5	ORC . . . . .	49
3.1.6	Generic Spark I/O . . . . .	51
3.1.7	Flat File / CSV . . . . .	53
3.1.8	Clipboard . . . . .	57
3.1.9	Excel . . . . .	59
3.1.10	JSON . . . . .	64
3.1.11	HTML . . . . .	67
3.1.12	SQL . . . . .	70
3.2	General functions . . . . .	72
3.2.1	Working with options . . . . .	72
3.2.2	Data manipulations and SQL . . . . .	74
3.2.3	Top-level missing data . . . . .	84
3.2.4	Top-level dealing with datetimelike . . . . .	91
3.3	Series . . . . .	95
3.3.1	Constructor . . . . .	95
3.3.2	Attributes . . . . .	101
3.3.3	Conversion . . . . .	107
3.3.4	Indexing, iteration . . . . .	109
3.3.5	Binary operator functions . . . . .	123
3.3.6	Function application, GroupBy & Window . . . . .	144
3.3.7	Computations / Descriptive Stats . . . . .	154

3.3.8	Reindexing / Selection / Label manipulation . . . . .	199
3.3.9	Missing data handling . . . . .	227
3.3.10	Reshaping, sorting, transposing . . . . .	236
3.3.11	Combining / joining / merging . . . . .	246
3.3.12	Time series-related . . . . .	252
3.3.13	Spark-related . . . . .	259
3.3.14	Accessors . . . . .	262
3.3.15	Date Time Handling . . . . .	262
3.3.16	String Handling . . . . .	277
3.3.17	Categorical accessor . . . . .	314
3.3.18	Plotting . . . . .	315
3.3.19	Serialization / IO / Conversion . . . . .	325
3.3.20	Koalas-specific . . . . .	339
3.4	DataFrame . . . . .	341
3.4.1	Constructor . . . . .	341
3.4.2	Attributes and underlying data . . . . .	349
3.4.3	Conversion . . . . .	355
3.4.4	Indexing, iteration . . . . .	361
3.4.5	Binary operator functions . . . . .	385
3.4.6	Function application, GroupBy & Window . . . . .	445
3.4.7	Computations / Descriptive Stats . . . . .	460
3.4.8	Reindexing / Selection / Label manipulation . . . . .	493
3.4.9	Missing data handling . . . . .	522
3.4.10	Reshaping, sorting, transposing . . . . .	530
3.4.11	Combining / joining / merging . . . . .	555
3.4.12	Time series-related . . . . .	564
3.4.13	Serialization / IO / Conversion . . . . .	568
3.4.14	Spark-related . . . . .	581
3.4.15	Plotting . . . . .	594
3.4.16	Koalas-specific . . . . .	604
3.5	Index objects . . . . .	611
3.5.1	Index . . . . .	611
3.5.2	Spark-related . . . . .	654
3.5.3	Numeric Index . . . . .	663
3.5.4	CategoricalIndex . . . . .	669
3.5.5	MultiIndex . . . . .	674
3.5.6	MultiIndex Spark-related . . . . .	710
3.5.7	DatetimeIndex . . . . .	713
3.6	Window . . . . .	729
3.6.1	Standard moving window functions . . . . .	729
3.6.2	Standard expanding window functions . . . . .	737
3.7	GroupBy . . . . .	741
3.7.1	Indexing, iteration . . . . .	741
3.7.2	Function application . . . . .	742
3.7.3	Computations / Descriptive Stats . . . . .	751
3.8	Machine Learning utilities . . . . .	778
3.8.1	MLflow . . . . .	778
3.9	Extensions . . . . .	780
3.9.1	Accessors . . . . .	780
<b>4</b>	<b>Development</b>	<b>785</b>
4.1	Contributing Guide . . . . .	785
4.1.1	Types of Contributions . . . . .	785
4.1.2	Step-by-step Guide For Code Contributions . . . . .	786

4.1.3	Environment Setup . . . . .	786
4.1.4	Running Tests . . . . .	787
4.1.5	Building Documentation . . . . .	787
4.1.6	Coding Conventions . . . . .	787
4.1.7	Doctest Conventions . . . . .	787
4.1.8	Release Guide . . . . .	788
4.2	Design Principles . . . . .	789
4.2.1	Be Pythonic . . . . .	789
4.2.2	Unify small data (pandas) API and big data (Spark) API, but pandas first . . . . .	789
4.2.3	Return Koalas data structure for big data, and pandas data structure for small data . . . . .	789
4.2.4	Provide discoverable APIs for common data science tasks . . . . .	790
4.2.5	Provide well documented APIs, with examples . . . . .	790
4.2.6	Guardrails to prevent users from shooting themselves in the foot . . . . .	790
4.2.7	Be a lean API layer and move fast . . . . .	791
4.2.8	High test coverage . . . . .	791
<b>5</b>	<b>Release Notes</b>	<b>793</b>
5.1	Version 1.8.2 . . . . .	793
5.1.1	Improvements and bug fixes . . . . .	793
5.2	Version 1.8.1 . . . . .	793
5.2.1	Improvements and bug fixes . . . . .	793
5.3	Version 1.8.0 . . . . .	793
5.3.1	Categorical type and <code>ExtensionDtype</code> . . . . .	794
5.3.2	Other new features, improvements and bug fixes . . . . .	794
5.4	Version 1.7.0 . . . . .	795
5.4.1	Switch the default plotting backend to Plotly . . . . .	795
5.4.2	Add <code>Int64Index</code> , <code>Float64Index</code> , <code>DatetimeIndex</code> . . . . .	795
5.4.3	Create Index from Series or Index objects . . . . .	797
5.4.4	Extension dtypes support . . . . .	797
5.4.5	Other new features, improvements and bug fixes . . . . .	798
5.5	Version 1.6.0 . . . . .	799
5.5.1	Improved Plotly backend support . . . . .	799
5.5.2	Operations between Series and Index . . . . .	800
5.5.3	Support setting to a <code>Series</code> via attribute access . . . . .	801
5.5.4	Other new features, improvements and bug fixes . . . . .	801
5.6	Version 1.5.0 . . . . .	802
5.6.1	Index operations support . . . . .	802
5.6.2	Other new features and improvements . . . . .	802
5.6.3	Other improvements and bug fixes . . . . .	803
5.7	Version 1.4.0 . . . . .	803
5.7.1	Better type support . . . . .	803
5.7.2	Return type annotations for major Koalas objects . . . . .	804
5.7.3	pandas 1.1.4 support . . . . .	805
5.7.4	Other new features and improvements . . . . .	805
5.7.5	Other improvements and bug fixes . . . . .	806
5.8	Version 1.3.0 . . . . .	806
5.8.1	pandas 1.1 support . . . . .	806
5.8.2	Support for non-string names . . . . .	806
5.8.3	Improve <code>distributed-sequence</code> default index . . . . .	807
5.8.4	Standardize binary operations between int and str columns . . . . .	807
5.8.5	Other new features and improvements . . . . .	807
5.8.6	Other improvements . . . . .	808
5.9	Version 1.2.0 . . . . .	809
5.9.1	Non-named Series support . . . . .	809

5.9.2	More stable “distributed-sequence” default index	809
5.9.3	Improve testing infrastructure	810
5.9.4	Other new features and improvements	810
5.9.5	Other improvements	810
5.10	Version 1.1.0	811
5.10.1	API extensions	811
5.10.2	Plotting backend	812
5.10.3	Koalas accessor	812
5.10.4	Other new features and improvements	814
5.10.5	Other improvements	814
5.11	Version 1.0.1	815
5.11.1	Critical bug fix	815
5.11.2	Other improvements	815
5.12	Version 1.0.0	815
5.12.1	Better pandas API coverage	815
5.12.2	Apache Spark 3.0	816
5.12.3	Python 3.8	816
5.12.4	Spark accessor	816
5.12.5	Better type hint support	817
5.12.6	Wider support of in-place update	817
5.12.7	Less restriction on <code>compute.ops_on_diff_frames</code>	818
5.12.8	Other new features and improvements	818
5.12.9	Backward Compatibility	820
5.13	Version 0.33.0	820
5.13.1	<code>apply</code> and <code>transform</code> Improvements	820
5.13.2	Spark Schema	821
5.13.3	GroupBy Improvements	822
5.13.4	Other new features and improvements	822
5.13.5	Other improvements	822
5.14	Version 0.32.0	822
5.14.1	Koalas documentation redesign	822
5.14.2	<code>transform_batch</code> and <code>apply_batch</code>	823
5.14.3	Other new features and improvements	823
5.14.4	Other improvements	824
5.15	Version 0.31.0	824
5.15.1	PyArrow>=0.15 support is back	824
5.15.2	Spark specific improvements	824
5.15.3	Other new features and improvements	825
5.15.4	Other improvements	825
5.16	Version 0.30.0	825
5.16.1	Slice column selection support in <code>loc</code>	825
5.16.2	Slice row selection support in <code>loc</code> for multi-index	826
5.16.3	Slice row selection support in <code>iloc</code>	826
5.16.4	Support of setting values via <code>loc</code> and <code>iloc</code> at Series	826
5.16.5	Other new features and improvements	827
5.16.6	Other improvements	827
5.17	Version 0.29.0	828
5.17.1	Slice support in <code>iloc</code>	828
5.17.2	Documentation	828
5.17.3	Other new features and improvements	829
5.17.4	Other improvements	829
5.18	Version 0.28.0	829
5.18.1	pandas 1.0 support	829
5.18.2	<code>map_in_pandas</code>	829

5.18.3	Standardize code style using Black	830
5.18.4	Other new features and improvements	830
5.18.5	Other improvements	830
5.19	Version 0.27.0	830
5.19.1	head ordering	830
5.19.2	GitHub Actions	831
5.19.3	Other new features and improvements	831
5.19.4	Other improvements	831
5.20	Version 0.26.0	832
5.20.1	iat indexer	832
5.20.2	Other new features and improvements	832
5.20.3	Other improvements	833
5.21	Version 0.25.0	833
5.21.1	loc and iloc indexers improvement	833
5.21.2	Other new features and improvements	834
5.21.3	Other improvements	834
5.22	Version 0.24.0	835
5.22.1	NumPy's universal function (ufunc) compatibility	835
5.22.2	Other new features and improvements	835
5.22.3	Other improvements	836
5.23	Version 0.23.0	836
5.23.1	NumPy's universal function (ufunc) compatibility	836
5.23.2	Other new features and improvements	837
5.23.3	Other improvements	837
5.24	Version 0.22.0	838
5.24.1	Enable Arrow 0.15.1+	838
5.24.2	Expanding and Rolling	838
5.24.3	Multi-index columns support	838
5.24.4	Documentation	838
5.24.5	Other new features and improvements	838
5.25	Version 0.21.0	839
5.25.1	Multi-index columns support	839
5.25.2	Documentation	839
5.25.3	Other new features and improvements	839
5.26	Version 0.20.0	840
5.26.1	Disable Arrow 0.15	840
5.26.2	Multi-index columns support	841
5.26.3	Other new features and improvements	841
5.27	Version 0.19.0	841
5.27.1	Koalas Logo	841
5.27.2	Documentation	842
5.27.3	Multi-index columns support	842
5.27.4	Plots	843
5.27.5	Other new features and improvements	843
5.28	Version 0.18.0	844
5.28.1	Multi-index columns support	844
5.28.2	Plots	844
5.28.3	Options	844
5.28.4	Other new features and improvements	845
5.28.5	Backward compatibility	845
5.29	Version 0.17.0	845
5.29.1	Options	845
5.29.2	Plots	846
5.29.3	Multi-index columns support	846



5.29.4	Other new features and improvements . . . . .	846
5.29.5	Backward compatibility . . . . .	847
5.30	Version 0.16.0 . . . . .	847
<b>Index</b>		<b>851</b>



**Warning:** DEPRECATED: Koalas supports Apache Spark 3.1 and below as it is [officially included to PySpark in Apache Spark 3.2](#). This repository is now in maintenance mode. For Apache Spark 3.2 and above, please use [PySpark](#) directly.

The Koalas project makes data scientists more productive when interacting with big data, by implementing the pandas DataFrame API on top of Apache Spark. pandas is the de facto standard (single-node) DataFrame implementation in Python, while Spark is the de facto standard for big data processing. With this package, you can:

- Be immediately productive with Spark, with no learning curve, if you are already familiar with pandas.
- Have a single codebase that works both with pandas (tests, smaller datasets) and with Spark (distributed datasets).

We would love to have you try it and give us feedback, through our [mailing lists](#) or [GitHub issues](#). Try the Koalas 10 minutes tutorial on a live Jupyter notebook [here](#). The initial launch can take up to several minutes.



## GETTING STARTED

### 1.1 Installation

Koalas requires PySpark so please make sure your PySpark is available.

To install Koalas, you can use:

- [Conda](#)
- [PyPI](#)
- *[Installation from source](#)*

To install PySpark, you can use:

- [Installation with the official release channel](#)
- [Conda](#)
- [PyPI](#)
- [Installation from source](#)

#### 1.1.1 Python version support

Officially Python 3.5 to 3.8.

---

**Note:** Koalas support for Python 3.5 is deprecated and will be dropped in the future release. At that point, existing Python 3.5 workflows that use Koalas will continue to work without modification, but Python 3.5 users will no longer get access to the latest Koalas features and bugfixes. We recommend that you upgrade to Python 3.6 or newer.

---

#### 1.1.2 Installing Koalas

##### Installing with Conda

First you will need [Conda](#) to be installed. After that, we should create a new conda environment. A conda environment is similar with a virtualenv that allows you to specify a specific version of Python and set of libraries. Run the following commands from a terminal window:

```
conda create --name koalas-dev-env
```

This will create a minimal environment with only Python installed in it. To put your self inside this environment run:

```
conda activate koalas-dev-env
```

The final step required is to install Koalas. This can be done with the following command:

```
conda install -c conda-forge koalas
```

To install a specific Koalas version:

```
conda install -c conda-forge koalas=1.3.0
```

### Installing from PyPI

Koalas can be installed via pip from [PyPI](#):

```
pip install koalas
```

### Installing from source

See the [Contribution Guide](#) for complete instructions.

## 1.1.3 Installing PySpark

### Installing with the official release channel

You can install PySpark by downloading a release in the [official release channel](#). Once you download the release, un-tar it first as below:

```
tar xzvf spark-2.4.4-bin-hadoop2.7.tgz
```

After that, make sure set SPARK\_HOME environment variable to indicate the directory you untar-ed:

```
cd spark-2.4.4-bin-hadoop2.7
export SPARK_HOME=`pwd`
```

Also, make sure your PYTHONPATH can find the PySpark and Py4J under \$SPARK\_HOME/python/lib:

```
export PYTHONPATH=$(ZIPS=("$SPARK_HOME"/python/lib/*.zip); IFS=:; echo "${ZIPS[*]}") :
↪ $PYTHONPATH
```

### Installing with Conda

PySpark can be installed via [Conda](#):

```
conda install -c conda-forge pyspark
```

## Installing with PyPI

PySpark can be installed via pip from [PyPI](#):

```
pip install pyspark
```

## Installing from source

To install PySpark from source, refer [Building Spark](#).

Likewise, make sure you set `SPARK_HOME` environment variable to the git-cloned directory, and your `PYTHONPATH` environment variable can find the PySpark and Py4J under `$SPARK_HOME/python/lib`:

```
export PYTHONPATH=$(ZIPS=("$SPARK_HOME"/python/lib/*.zip); IFS=:; echo "${ZIPS[*]}"):
↪ $PYTHONPATH
```

### 1.1.4 Dependencies

Package	Required version
<i>pandas</i>	>=0.23.2
<i>pyspark</i>	>=2.4.0
<i>pyarrow</i>	>=0.10
<i>numpy</i>	>=1.14

### Optional dependencies

Package	Required version
<i>mlflow</i>	>=1.0
<i>plotly</i>	>=4.8
<i>matplotlib</i>	>=3.0.0,<3.3.0

## 1.2 10 minutes to Koalas

This is a short introduction to Koalas, geared mainly for new users. This notebook shows you some key differences between pandas and Koalas. You can run this examples by yourself on a live notebook [here](#). For Databricks Runtime, you can import and run the [current .ipynb file](#) out of the box. Try it on [Databricks Community Edition](#) for free.

Customarily, we import Koalas as follows:

```
[1]: import pandas as pd
import numpy as np
import databricks.koalas as ks
from pyspark.sql import SparkSession
```

## 1.2.1 Object Creation

Creating a Koalas Series by passing a list of values, letting Koalas create a default integer index:

```
[2]: s = ks.Series([1, 3, 5, np.nan, 6, 8])
```

```
[3]: s
```

```
[3]: 0    1.0
     1    3.0
     2    5.0
     3    NaN
     4    6.0
     5    8.0
     dtype: float64
```

Creating a Koalas DataFrame by passing a dict of objects that can be converted to series-like.

```
[4]: kdf = ks.DataFrame(
      {'a': [1, 2, 3, 4, 5, 6],
       'b': [100, 200, 300, 400, 500, 600],
       'c': ["one", "two", "three", "four", "five", "six"]},
      index=[10, 20, 30, 40, 50, 60])
```

```
[5]: kdf
```

```
[5]:   a    b    c
     10  1  100  one
     20  2  200  two
     30  3  300 three
     40  4  400  four
     50  5  500  five
     60  6  600  six
```

Creating a pandas DataFrame by passing a numpy array, with a datetime index and labeled columns:

```
[6]: dates = pd.date_range('20130101', periods=6)
```

```
[7]: dates
```

```
[7]: DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
                  '2013-01-05', '2013-01-06'],
                  dtype='datetime64[ns]', freq='D')
```

```
[8]: pdf = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list('ABCD'))
```

```
[9]: pdf
```

```
[9]:              A         B         C         D
2013-01-01 -0.621429  1.515041 -1.735483 -1.235009
2013-01-02  0.844961 -0.999771  0.108356  0.109456
2013-01-03  1.343862 -1.257980  0.099766 -0.137677
2013-01-04  3.001767 -0.208167 -1.059449  0.312599
2013-01-05 -0.035864  0.312126  0.252281  0.627551
2013-01-06 -1.200404  0.276134 -0.344308 -0.367934
```

Now, this pandas DataFrame can be converted to a Koalas DataFrame



```
[10]: kdf = ks.from_pandas(pdf)
```

```
[11]: type(kdf)
```

```
[11]: databricks.koalas.frame.DataFrame
```

It looks and behaves the same as a pandas DataFrame though

```
[12]: kdf
```

```
[12]:
```

	A	B	C	D
2013-01-01	-0.621429	1.515041	-1.735483	-1.235009
2013-01-02	0.844961	-0.999771	0.108356	0.109456
2013-01-03	1.343862	-1.257980	0.099766	-0.137677
2013-01-04	3.001767	-0.208167	-1.059449	0.312599
2013-01-05	-0.035864	0.312126	0.252281	0.627551
2013-01-06	-1.200404	0.276134	-0.344308	-0.367934

Also, it is possible to create a Koalas DataFrame from Spark DataFrame.

Creating a Spark DataFrame from pandas DataFrame

```
[13]: spark = SparkSession.builder.getOrCreate()
```

```
[14]: sdf = spark.createDataFrame(pdf)
```

```
[15]: sdf.show()
```

```
+-----+-----+-----+-----+
|          A|          B|          C|          D|
+-----+-----+-----+-----+
| -0.6214290839748133| 1.5150410562536945| -1.7354827055737831| -1.2350091172431052|
| 0.8449607212376394| -0.9997705636655247| 0.10835607649858589| 0.1094555359929294|
| 1.3438622379103737| -1.2579798113362755| 0.0997664833965215| -0.13767658889070905|
| 3.001767403315059| -0.20816676142436616| -1.0594485090898984| 0.31259853367492724|
| -0.03586387305407219| 0.3121259401964947| 0.2522808041799677| 0.6275512901423211|
| -1.2004042904971255| 0.27613400857508563| -0.34430818441482375| -0.36793440398703187|
+-----+-----+-----+-----+
```

Creating Koalas DataFrame from Spark DataFrame. `to_koalas()` is automatically attached to Spark DataFrame and available as an API when Koalas is imported.

```
[16]: kdf = sdf.to_koalas()
```

```
[17]: kdf
```

```
[17]:
```

	A	B	C	D
0	-0.621429	1.515041	-1.735483	-1.235009
1	0.844961	-0.999771	0.108356	0.109456
2	1.343862	-1.257980	0.099766	-0.137677
3	3.001767	-0.208167	-1.059449	0.312599
4	-0.035864	0.312126	0.252281	0.627551
5	-1.200404	0.276134	-0.344308	-0.367934

Having specific `dtypes`. Types that are common to both Spark and pandas are currently supported.

```
[18]: kdf.dtypes
[18]: A      float64
      B      float64
      C      float64
      D      float64
      dtype: object
```

## 1.2.2 Viewing Data

See the [API Reference](#).

See the top rows of the frame. The results may not be the same as pandas though: unlike pandas, the data in a Spark dataframe is not *ordered*, it has no intrinsic notion of index. When asked for the head of a dataframe, Spark will just take the requested number of rows from a partition. Do not rely on it to return specific rows, use `.loc` or `iloc` instead.

```
[19]: kdf.head()
[19]:
```

	A	B	C	D
0	-0.621429	1.515041	-1.735483	-1.235009
1	0.844961	-0.999771	0.108356	0.109456
2	1.343862	-1.257980	0.099766	-0.137677
3	3.001767	-0.208167	-1.059449	0.312599
4	-0.035864	0.312126	0.252281	0.627551

Display the index, columns, and the underlying numpy data.

You can also retrieve the index; the index column can be ascribed to a DataFrame, see later

```
[20]: kdf.index
[20]: Int64Index([0, 1, 2, 3, 4, 5], dtype='int64')
```

```
[21]: kdf.columns
[21]: Index(['A', 'B', 'C', 'D'], dtype='object')
```

```
[22]: kdf.to_numpy()
[22]: array([[ -0.62142908,  1.51504106, -1.73548271, -1.23500912],
        [ 0.84496072, -0.99977056,  0.10835608,  0.10945554],
        [ 1.34386224, -1.25797981,  0.09976648, -0.13767659],
        [ 3.0017674 , -0.20816676, -1.05944851,  0.31259853],
        [-0.03586387,  0.31212594,  0.2522808 ,  0.62755129],
        [-1.20040429,  0.27613401, -0.34430818, -0.3679344 ]])
```

Describe shows a quick statistic summary of your data

```
[23]: kdf.describe()
[23]:
```

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	0.555482	-0.060436	-0.446473	-0.115169
std	1.517076	1.007223	0.792741	0.648616
min	-1.200404	-1.257980	-1.735483	-1.235009
25%	-0.621429	-0.999771	-1.059449	-0.367934
50%	-0.035864	-0.208167	-0.344308	-0.137677

(continues on next page)

(continued from previous page)

75%	1.343862	0.312126	0.108356	0.312599
max	3.001767	1.515041	0.252281	0.627551

Transposing your data

```
[24]: kdf.T
```

```
[24]:
```

	0	1	2	3	4	5
A	-0.621429	0.844961	1.343862	3.001767	-0.035864	-1.200404
B	1.515041	-0.999771	-1.257980	-0.208167	0.312126	0.276134
C	-1.735483	0.108356	0.099766	-1.059449	0.252281	-0.344308
D	-1.235009	0.109456	-0.137677	0.312599	0.627551	-0.367934

Sorting by its index

```
[25]: kdf.sort_index(ascending=False)
```

```
[25]:
```

	A	B	C	D
5	-1.200404	0.276134	-0.344308	-0.367934
4	-0.035864	0.312126	0.252281	0.627551
3	3.001767	-0.208167	-1.059449	0.312599
2	1.343862	-1.257980	0.099766	-0.137677
1	0.844961	-0.999771	0.108356	0.109456
0	-0.621429	1.515041	-1.735483	-1.235009

Sorting by value

```
[26]: kdf.sort_values(by='B')
```

```
[26]:
```

	A	B	C	D
2	1.343862	-1.257980	0.099766	-0.137677
1	0.844961	-0.999771	0.108356	0.109456
3	3.001767	-0.208167	-1.059449	0.312599
5	-1.200404	0.276134	-0.344308	-0.367934
4	-0.035864	0.312126	0.252281	0.627551
0	-0.621429	1.515041	-1.735483	-1.235009

## 1.2.3 Missing Data

Koalas primarily uses the value `np.nan` to represent missing data. It is by default not included in computations.

```
[27]: pdf1 = pdf.reindex(index=dates[0:4], columns=list(pdf.columns) + ['E'])
```

```
[28]: pdf1.loc[dates[0]:dates[1], 'E'] = 1
```

```
[29]: kdf1 = ks.from_pandas(pdf1)
```

```
[30]: kdf1
```

```
[30]:
```

	A	B	C	D	E
2013-01-01	-0.621429	1.515041	-1.735483	-1.235009	1.0
2013-01-02	0.844961	-0.999771	0.108356	0.109456	1.0
2013-01-03	1.343862	-1.257980	0.099766	-0.137677	NaN
2013-01-04	3.001767	-0.208167	-1.059449	0.312599	NaN

To drop any rows that have missing data.

```
[31]: kdf1.dropna(how='any')
```

```
[31]:
```

	A	B	C	D	E
2013-01-01	-0.621429	1.515041	-1.735483	-1.235009	1.0
2013-01-02	0.844961	-0.999771	0.108356	0.109456	1.0

Filling missing data.

```
[32]: kdf1.fillna(value=5)
```

```
[32]:
```

	A	B	C	D	E
2013-01-01	-0.621429	1.515041	-1.735483	-1.235009	1.0
2013-01-02	0.844961	-0.999771	0.108356	0.109456	1.0
2013-01-03	1.343862	-1.257980	0.099766	-0.137677	5.0
2013-01-04	3.001767	-0.208167	-1.059449	0.312599	5.0

## 1.2.4 Operations

### Stats

Operations in general exclude missing data.

Performing a descriptive statistic:

```
[33]: kdf.mean()
```

```
[33]: A    0.555482
      B   -0.060436
      C   -0.446473
      D   -0.115169
      dtype: float64
```

### Spark Configurations

Various configurations in PySpark could be applied internally in Koalas. For example, you can enable Arrow optimization to hugely speed up internal pandas conversion. See [PySpark Usage Guide for Pandas with Apache Arrow](#).

```
[34]: prev = spark.conf.get("spark.sql.execution.arrow.enabled") # Keep its default value.
      ks.set_option("compute.default_index_type", "distributed") # Use default index_
      ↪ prevent overhead.
      import warnings
      warnings.filterwarnings("ignore") # Ignore warnings coming from Arrow optimizations.
```

```
[35]: spark.conf.set("spark.sql.execution.arrow.enabled", True)
      %timeit ks.range(300000).to_pandas()

311 ms ± 30.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
[36]: spark.conf.set("spark.sql.execution.arrow.enabled", False)
      %timeit ks.range(300000).to_pandas()

1.25 s ± 29.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
[37]: ks.reset_option("compute.default_index_type")
      spark.conf.set("spark.sql.execution.arrow.enabled", prev) # Set its default value_
      ↪ back.
```

## 1.2.5 Grouping

By “group by” we are referring to a process involving one or more of the following steps:

- Splitting the data into groups based on some criteria
- Applying a function to each group independently
- Combining the results into a data structure

```
[38]: kdf = ks.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
                               'foo', 'bar', 'foo', 'foo'],
                          'B': ['one', 'one', 'two', 'three',
                               'two', 'two', 'one', 'three'],
                          'C': np.random.randn(8),
                          'D': np.random.randn(8)})
```

```
[39]: kdf
```

```
[39]:
```

	A	B	C	D
0	foo	one	0.392094	-0.197885
1	bar	one	0.397240	0.768301
2	foo	two	-1.683135	-0.210606
3	bar	three	-1.776986	-0.092022
4	foo	two	-0.499332	0.463287
5	bar	two	0.386921	1.995358
6	foo	one	-0.514731	1.042816
7	foo	three	0.194186	1.745033

Grouping and then applying the `sum()` function to the resulting groups.

```
[40]: kdf.groupby('A').sum()
```

```
[40]:
```

	C	D
A		
bar	-0.992825	2.671637
foo	-2.110918	2.842644

Grouping by multiple columns forms a hierarchical index, and again we can apply the `sum` function.

```
[41]: kdf.groupby(['A', 'B']).sum()
```

```
[41]:
```

		C	D
A	B		
foo	one	-0.122637	0.844931
	two	-2.182467	0.252681
bar	three	-1.776986	-0.092022
foo	three	0.194186	1.745033
bar	two	0.386921	1.995358
	one	0.397240	0.768301

## 1.2.6 Plotting

See the Plotting docs.

```
[42]: pser = pd.Series(np.random.randn(1000),  
                      index=pd.date_range('1/1/2000', periods=1000))
```

```
[43]: kser = ks.Series(pser)
```

```
[44]: kser = kser.cummax()
```

```
[45]: kser.plot()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

On a DataFrame, the plot() method is a convenience to plot all of the columns with labels:

```
[46]: pdf = pd.DataFrame(np.random.randn(1000, 4), index=pser.index,  
                        columns=['A', 'B', 'C', 'D'])
```

```
[47]: kdf = ks.from_pandas(pdf)
```

```
[48]: kdf = kdf.cummax()
```

```
[49]: kdf.plot()
```

Data type cannot be displayed: application/vnd.plotly.v1+json, text/html

## 1.2.7 Getting data in/out

See the Input/Output docs.

### CSV

CSV is straightforward and easy to use. See [here](#) to write a CSV file and [here](#) to read a CSV file.

```
[50]: kdf.to_csv('foo.csv')  
ks.read_csv('foo.csv').head(10)
```

```
[50]:
```

	A	B	C	D
0	-0.821342	-0.325142	0.904636	-0.925984
1	1.498758	0.045747	0.904636	0.726606
2	1.498758	0.045747	0.904636	0.726606
3	1.498758	1.534086	0.904636	0.726606
4	1.498758	1.534086	0.904636	0.726606
5	1.498758	1.534086	0.904636	0.726606
6	1.498758	1.534086	0.904636	0.726606
7	1.498758	1.534086	0.904636	0.856176
8	1.498758	1.534086	0.904636	0.856176
9	1.498758	1.534086	0.904636	1.532448

## Parquet

Parquet is an efficient and compact file format to read and write faster. See [here](#) to write a Parquet file and [here](#) to read a Parquet file.

```
[51]: kdf.to_parquet('bar.parquet')
ks.read_parquet('bar.parquet').head(10)
```

```
[51]:
```

	A	B	C	D
0	-0.821342	-0.325142	0.904636	-0.925984
1	1.498758	0.045747	0.904636	0.726606
2	1.498758	0.045747	0.904636	0.726606
3	1.498758	1.534086	0.904636	0.726606
4	1.498758	1.534086	0.904636	0.726606
5	1.498758	1.534086	0.904636	0.726606
6	1.498758	1.534086	0.904636	0.726606
7	1.498758	1.534086	0.904636	0.856176
8	1.498758	1.534086	0.904636	0.856176
9	1.498758	1.534086	0.904636	1.532448

## Spark IO

In addition, Koalas fully support Spark's various datasources such as ORC and an external datasource. See [here](#) to write it to the specified datasource and [here](#) to read it from the datasource.

```
[52]: kdf.to_spark_io('zoo.orc', format="orc")
ks.read_spark_io('zoo.orc', format="orc").head(10)
```

```
[52]:
```

	A	B	C	D
0	-0.821342	-0.325142	0.904636	-0.925984
1	1.498758	0.045747	0.904636	0.726606
2	1.498758	0.045747	0.904636	0.726606
3	1.498758	1.534086	0.904636	0.726606
4	1.498758	1.534086	0.904636	0.726606
5	1.498758	1.534086	0.904636	0.726606
6	1.498758	1.534086	0.904636	0.726606
7	1.498758	1.534086	0.904636	0.856176
8	1.498758	1.534086	0.904636	0.856176
9	1.498758	1.534086	0.904636	1.532448

## 1.3 Koalas Talks and Blogs

### 1.3.1 Blog Posts

- [Interoperability between Koalas and Apache Spark \(Aug 11, 2020\)](#)
- [Introducing Koalas 1.0 \(Jun 24, 2020\)](#)
- [10 Minutes from pandas to Koalas on Apache Spark \(Mar 31, 2020\)](#)
- [Guest Blog: How Virgin Hyperloop One Reduced Processing Time from Hours to Minutes with Koalas \(Aug 22, 2019\)](#)
- [Koalas: Easy Transition from pandas to Apache Spark \(Apr 24, 2019\)](#)

### **1.3.2 Data + AI Summit 2020 EUROPE (Nov 18-19, 2020)**

Project Zen: Making Spark Pythonic

Koalas: Interoperability Between Koalas and Apache Spark

### **1.3.3 Spark + AI Summit 2020 (Jun 24, 2020)**

Introducing Apache Spark 3.0: A retrospective of the Last 10 Years, and a Look Forward to the Next 10 Years to Come.

Koalas: Making an Easy Transition from Pandas to Apache Spark

Koalas: Pandas on Apache Spark

### **1.3.4 Webinar @ Databricks (Mar 27, 2020)**

Reducing Time-To-Insight for Virgin Hyperloop's Data

### **1.3.5 PyData New York 2019 (Nov 4, 2019)**

Pandas vs Koalas: The Ultimate Showdown

### **1.3.6 Spark + AI Summit Europe 2019 (Oct 16, 2019)**

New Developments in the Open Source Ecosystem: Apache Spark 3.0, Delta Lake, and Koalas

Koalas: Making an Easy Transition from Pandas to Apache Spark

Koalas: Pandas on Apache Spark

### **1.3.7 PyBay 2019 (Aug 17, 2019)**

Koalas Easy Transition from pandas to Apache Spark

### **1.3.8 Spark + AI Summit 2019 (Apr 24, 2019)**

Official Announcement of Koalas Open Source Project



## 2.1 Options and settings

Koalas has an options system that lets you customize some aspects of its behaviour, display-related options being those the user is most likely to adjust.

Options have a full “dotted-style”, case-insensitive name (e.g. `display.max_rows`). You can get/set options directly as attributes of the top-level `options` attribute:

```
>>> import databricks.koalas as ks
>>> ks.options.display.max_rows
1000
>>> ks.options.display.max_rows = 10
>>> ks.options.display.max_rows
10
```

The API is composed of 3 relevant functions, available directly from the `koalas` namespace:

- `get_option()` / `set_option()` - get/set the value of a single option.
- `reset_option()` - reset one or more options to their default value.

**Note:** Developers can check out `databricks/koalas/config.py` for more information.

```
>>> import databricks.koalas as ks
>>> ks.get_option("display.max_rows")
1000
>>> ks.set_option("display.max_rows", 101)
>>> ks.get_option("display.max_rows")
101
```

### 2.1.1 Getting and setting options

As described above, `get_option()` and `set_option()` are available from the `koalas` namespace. To change an option, call `set_option('option name', new_value)`.

```
>>> import databricks.koalas as ks
>>> ks.get_option('compute.max_rows')
1000
>>> ks.set_option('compute.max_rows', 2000)
>>> ks.get_option('compute.max_rows')
2000
```

All options also have a default value, and you can use `reset_option` to do just that:

```
>>> import databricks.koalas as ks
>>> ks.reset_option("display.max_rows")
```

```
>>> import databricks.koalas as ks
>>> ks.get_option("display.max_rows")
1000
>>> ks.set_option("display.max_rows", 999)
>>> ks.get_option("display.max_rows")
999
>>> ks.reset_option("display.max_rows")
>>> ks.get_option("display.max_rows")
1000
```

option\_context context manager has been exposed through the top-level API, allowing you to execute code with given option values. Option values are restored automatically when you exit the *with* block:

```
>>> with ks.option_context("display.max_rows", 10, "compute.max_rows", 5):
...     print(ks.get_option("display.max_rows"))
...     print(ks.get_option("compute.max_rows"))
10
5
>>> print(ks.get_option("display.max_rows"))
>>> print(ks.get_option("compute.max_rows"))
1000
1000
```

## 2.1.2 Operations on different DataFrames

Koalas disallows the operations on different DataFrames (or Series) by default to prevent expensive operations. It internally performs a join operation which can be expensive in general.

This can be enabled by setting `compute.ops_on_diff_frames` to `True` to allow such cases. See the examples below.

```
>>> import databricks.koalas as ks
>>> ks.set_option('compute.ops_on_diff_frames', True)
>>> kdf1 = ks.range(5)
>>> kdf2 = ks.DataFrame({'id': [5, 4, 3]})
>>> (kdf1 - kdf2).sort_index()
   id
0 -5.0
1 -3.0
2 -1.0
3  NaN
4  NaN
>>> ks.reset_option('compute.ops_on_diff_frames')
```

```
>>> import databricks.koalas as ks
>>> ks.set_option('compute.ops_on_diff_frames', True)
>>> kdf = ks.range(5)
>>> kser_a = ks.Series([1, 2, 3, 4])
>>> # 'kser_a' is not from 'kdf' DataFrame. So it is considered as a Series not from
↳ 'kdf'.
>>> kdf['new_col'] = kser_a
>>> kdf
   id  new_col
0    0        1
1    1        2
2    2        3
3    3        4
4    4        5
```

(continues on next page)

(continued from previous page)

```

0    0    1.0
1    1    2.0
3    3    4.0
2    2    3.0
4    4    NaN
>>> ks.reset_option('compute.ops_on_diff_frames')

```

### 2.1.3 Default Index type

In Koalas, the default index is used in several cases, for instance, when Spark DataFrame is converted into Koalas DataFrame. In this case, internally Koalas attaches a default index into Koalas DataFrame.

There are several types of the default index that can be configured by `compute.default_index_type` as below:

**sequence:** It implements a sequence that increases one by one, by PySpark's Window function without specifying partition. Therefore, it can end up with whole partition in single node. This index type should be avoided when the data is large. This is default. See the example below:

```

>>> import databricks.koalas as ks
>>> ks.set_option('compute.default_index_type', 'sequence')
>>> kdf = ks.range(3)
>>> ks.reset_option('compute.default_index_type')
>>> kdf.index
Int64Index([0, 1, 2], dtype='int64')

```

This is conceptually equivalent to the PySpark example as below:

```

>>> from pyspark.sql import functions as F, Window
>>> import databricks.koalas as ks
>>> spark_df = ks.range(3).to_spark()
>>> sequential_index = F.row_number().over(
...     Window.orderBy(F.monotonically_increasing_id().asc())) - 1
>>> spark_df.select(sequential_index).rdd.map(lambda r: r[0]).collect()
[0, 1, 2]

```

**distributed-sequence:** It implements a sequence that increases one by one, by group-by and group-map approach in a distributed manner. It still generates the sequential index globally. If the default index must be the sequence in a large dataset, this index has to be used. Note that if more data are added to the data source after creating this index, then it does not guarantee the sequential index. See the example below:

```

>>> import databricks.koalas as ks
>>> ks.set_option('compute.default_index_type', 'distributed-sequence')
>>> kdf = ks.range(3)
>>> ks.reset_option('compute.default_index_type')
>>> kdf.index
Int64Index([0, 1, 2], dtype='int64')

```

This is conceptually equivalent to the PySpark example as below:

```

>>> import databricks.koalas as ks
>>> spark_df = ks.range(3).to_spark()
>>> spark_df.rdd.zipWithIndex().map(lambda p: p[1]).collect()
[0, 1, 2]

```

**distributed:** It implements a monotonically increasing sequence simply by using PySpark's `monotonically_increasing_id` function in a fully distributed manner. The values are indeterministic. If the index does not

have to be a sequence that increases one by one, this index should be used. Performance-wise, this index almost does not have any penalty comparing to other index types. See the example below:

```
>>> import databricks.koalas as ks
>>> ks.set_option('compute.default_index_type', 'distributed')
>>> kdf = ks.range(3)
>>> ks.reset_option('compute.default_index_type')
>>> kdf.index
Int64Index([25769803776, 60129542144, 94489280512], dtype='int64')
```

This is conceptually equivalent to the PySpark example as below:

```
>>> from pyspark.sql import functions as F
>>> import databricks.koalas as ks
>>> spark_df = ks.range(3).to_spark()
>>> spark_df.select(F.monotonically_increasing_id()) \
...     .rdd.map(lambda r: r[0]).collect()
[25769803776, 60129542144, 94489280512]
```

**Warning:** It is very unlikely for this type of index to be used for computing two different dataframes because it is not guaranteed to have the same indexes in two dataframes. If you use this default index and turn on *compute.ops\_on\_diff\_frames*, the result from the operations between two different DataFrames will likely be an unexpected output due to the indeterministic index values.

## 2.1.4 Available options

Option	Default	Description
display.max_rows	1000	This sets the maximum number of rows Koalas should output when printing out various output. For example, this value determines the number of rows to be shown at the repr() in a dataframe. Set <i>None</i> to unlimit the input length. Default is 1000.
compute.max_rows	1000	'compute.max_rows' sets the limit of the current Koalas DataFrame. Set <i>None</i> to unlimit the input length. When the limit is set, it is executed by the shortcut by collecting the data into the driver, and then using the pandas API. If the limit is unset, the operation is executed by PySpark. Default is 1000.
compute.shortcut_limit	1000	'compute.shortcut_limit' sets the limit for a shortcut. It computes specified number of rows and reuse its schema. When the dataframe length is larger than this limit, Koalas uses PySpark to compute.
compute.ops_on_diff_frames	False	This determines whether or not to operate between two different dataframes. For example, 'compute.ops_on_diff_frames' function internally performs a join operation which can be expensive in general. So, if <i>compute.ops_on_diff_frames</i> variable is not True, that method throws an exception.
compute.default_index_type	'sequence'	This sets the default index type: sequence, distributed and distributed-sequence.
compute.ordered_head	False	'compute.ordered_head' sets whether or not to operate head with natural ordering. Koalas does not guarantee the row ordering so <i>head</i> could return some rows from distributed partitions. If 'compute.ordered_head' is set to True, Koalas performs natural ordering beforehand, but it will cause a performance overhead.
plotting.max_rows	1000	'plotting.max_rows' sets the visual limit on top-n- based plots such as <i>plot.bar</i> and <i>plot.pie</i> . If it is set to 1000, the first 1000 data points will be used for plotting. Default is 1000.
plotting.sample_ratio	None	'plotting.sample_ratio' sets the proportion of data that will be plotted for sample-based plots such as <i>plot.line</i> and <i>plot.area</i> . This option defaults to 'plotting.max_rows' option.
plotting.backend	'plotly'	Backend to use for plotting. Default is plotly. Supports any package that has a top-level <i>.plot</i> method. Known options are: [matplotlib, plotly].

## 2.2 Working with pandas and PySpark

Users from pandas and/or PySpark face API compatibility issue sometimes when they work with Koalas. Since Koalas does not target 100% compatibility of both pandas and PySpark, users need to do some workaround to port their pandas and/or PySpark codes or get familiar with Koalas in this case. This page aims to describe it.

### 2.2.1 pandas

pandas users can access to full pandas APIs by calling `DataFrame.to_pandas()`. Koalas DataFrame and pandas DataFrame are similar. However, the former is distributed and the latter is in a single machine. When converting to each other, the data is transferred between multiple machines and the single client machine.

For example, if you need to call `pandas_df.values` of pandas DataFrame, you can do as below:

```
>>> import databricks.koalas as ks
>>>
>>> kdf = ks.range(10)
>>> pdf = kdf.to_pandas()
>>> pdf.values
array([[0],
```

(continues on next page)

(continued from previous page)

```
[1],  
[2],  
[3],  
[4],  
[5],  
[6],  
[7],  
[8],  
[9]])
```

pandas DataFrame can be a Koalas DataFrame easily as below:

```
>>> ks.from_pandas(pdf)  
   id  
0    0  
1    1  
2    2  
3    3  
4    4  
5    5  
6    6  
7    7  
8    8  
9    9
```

Note that converting Koalas DataFrame to pandas requires to collect all the data into the client machine; therefore, if possible, it is recommended to use Koalas or PySpark APIs instead.

## 2.2.2 PySpark

PySpark users can access to full PySpark APIs by calling `DataFrame.to_spark()`. Koalas DataFrame and Spark DataFrame are virtually interchangeable.

For example, if you need to call `spark_df.filter(...)` of Spark DataFrame, you can do as below:

```
>>> import databricks.koalas as ks  
>>>  
>>> kdf = ks.range(10)  
>>> sdf = kdf.to_spark().filter("id > 5")  
>>> sdf.show()  
+----+  
| id |  
+----+  
|  6 |  
|  7 |  
|  8 |  
|  9 |  
+----+
```

Spark DataFrame can be a Koalas DataFrame easily as below:

```
>>> sdf.to_koalas()  
   id  
0    6  
1    7
```

(continues on next page)

(continued from previous page)

```
2    8
3    9
```

However, note that it requires to create new default index in case Koalas DataFrame is created from Spark DataFrame. See [Default Index Type](#). In order to avoid this overhead, specify the column to use as an index when possible.

```
>>> # Create a Koalas DataFrame with an explicit index.
... kdf = ks.DataFrame({'id': range(10)}, index=range(10))
>>> # Keep the explicit index.
... sdf = kdf.to_spark(index_col='index')
>>> # Call Spark APIs
... sdf = sdf.filter("id > 5")
>>> # Uses the explicit index to avoid to create default index.
... sdf.to_koalas(index_col='index')
      id
index
6      6
7      7
8      8
9      9
```

## 2.3 Transform and apply a function

There are many APIs that allow users to apply a function against Koalas DataFrame such as `DataFrame.transform()`, `DataFrame.apply()`, `DataFrame.koalas.transform_batch()`, `DataFrame.koalas.apply_batch()`, `Series.koalas.transform_batch()`, etc. Each has a distinct purpose and works differently internally. This section describes the differences among them where users are confused often.

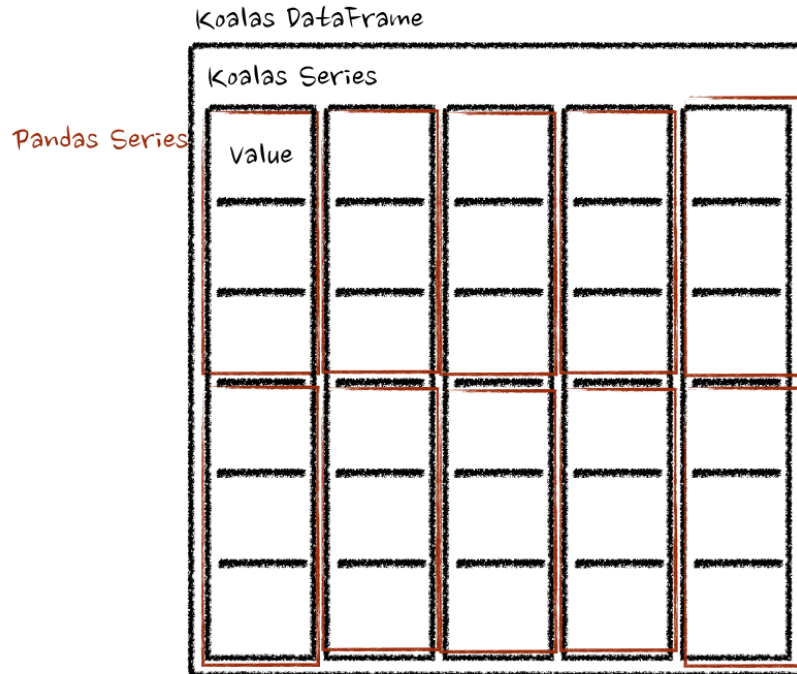
### 2.3.1 transform and apply

The main difference between `DataFrame.transform()` and `DataFrame.apply()` is that the former requires to return the same length of the input and the latter does not require this. See the example below:

```
>>> kdf = ks.DataFrame({'a': [1,2,3], 'b':[4,5,6]})
>>> def pandas_plus(pser):
...     return pser + 1 # should always return the same length as input.
...
>>> kdf.transform(pandas_plus)
```

```
>>> kdf = ks.DataFrame({'a': [1,2,3], 'b':[5,6,7]})
>>> def pandas_plus(pser):
...     return pser[pser % 2 == 1] # allows an arbitrary length
...
>>> kdf.apply(pandas_plus)
```

In this case, each function takes a pandas Series, and Koalas computes the functions in a distributed manner as below.

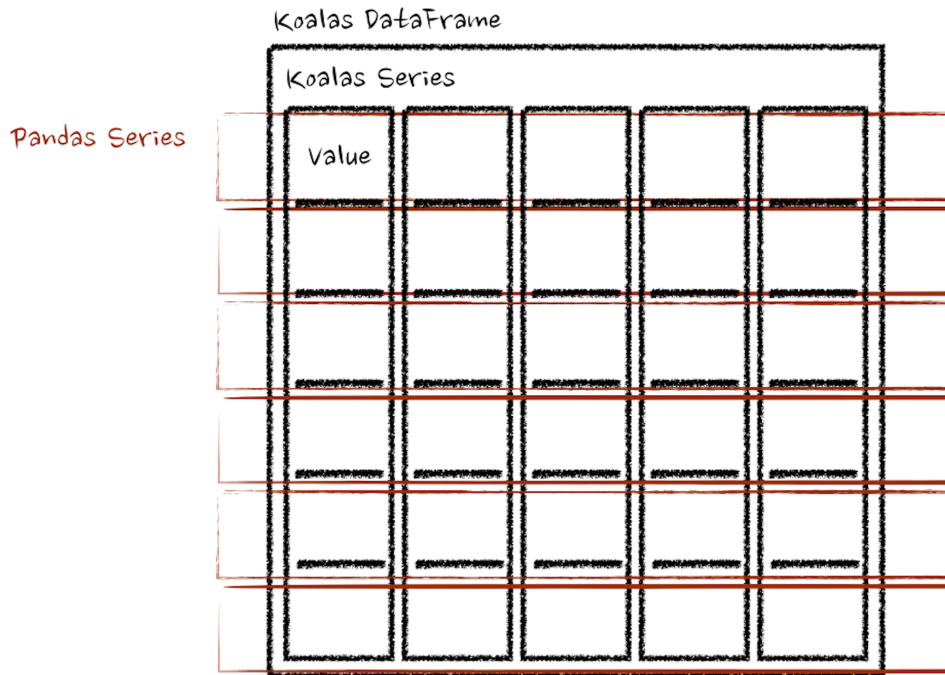


In case of 'column' axis, the function takes each row as a pandas Series.

```
>>> kdf = ks.DataFrame({'a': [1,2,3], 'b': [4,5,6]})
>>> def pandas_plus(pser):
...     return sum(pser) # allows an arbitrary length
...
>>> kdf.apply(pandas_plus, axis='columns')
```

The example above calculates the summation of each row as a pandas Series. See below:





In the examples above, the type hints were not used for simplicity but it is encouraged to use to avoid performance penalty. Please refer the API documentations.

### 2.3.2 `koalas.transform_batch` and `koalas.apply_batch`

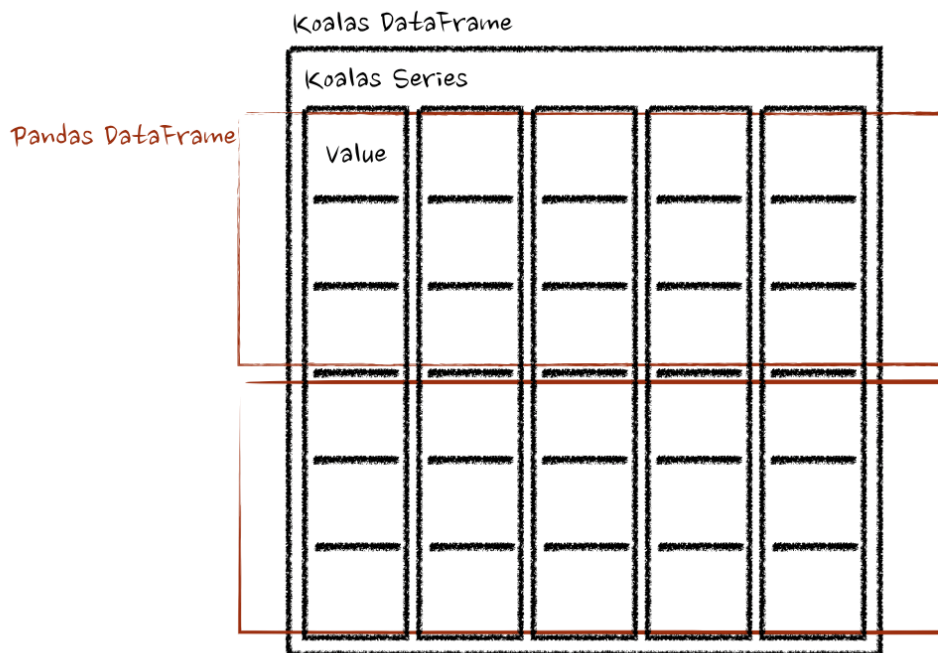
In `DataFrame.koalas.transform_batch()`, `DataFrame.koalas.apply_batch()`, `Series.koalas.transform_batch()`, etc., the batch postfix means each chunk in Koalas DataFrame or Series. The APIs slice the Koalas DataFrame or Series, and then applies the given function with pandas DataFrame or Series as input and output. See the examples below:

```
>>> kdf = ks.DataFrame({'a': [1,2,3], 'b':[4,5,6]})
>>> def pandas_plus(pdf):
...     return pdf + 1 # should always return the same length as input.
...
>>> kdf.koalas.transform_batch(pandas_plus)
```

```
>>> kdf = ks.DataFrame({'a': [1,2,3], 'b':[4,5,6]})
>>> def pandas_plus(pdf):
...     return pdf[pdf.a > 1] # allow arbitrary length
...
>>> kdf.koalas.apply_batch(pandas_plus)
```

The functions in both examples take a pandas DataFrame as a chunk of Koalas DataFrame, and output a pandas DataFrame. Koalas combines the pandas DataFrames as a Koalas DataFrame.

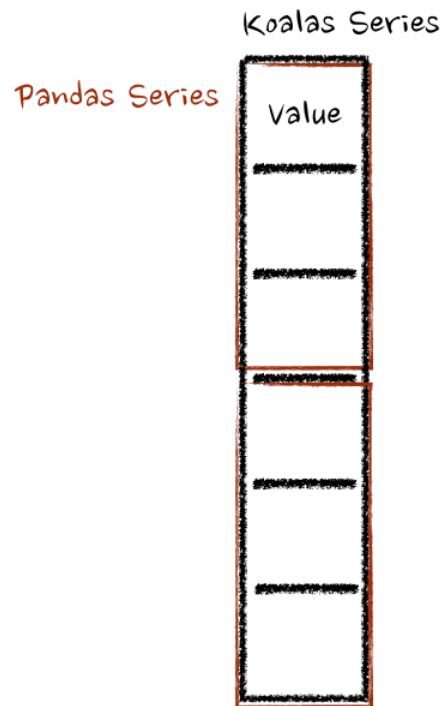
Note that `DataFrame.koalas.transform_batch()` has the length restriction - the length of input and output should be the same whereas `DataFrame.koalas.apply_batch()` does not. However, it is important to know that the output belongs to the same DataFrame when `DataFrame.koalas.transform_batch()` returns a Series, and you can avoid a shuffle by the operations between different DataFrames. In case of `DataFrame.koalas.apply_batch()`, its output is always treated that it belongs to a new different DataFrame. See also [Operations on different DataFrames](#) for more details.



In case of `Series.koalas.transform_batch()`, it is also similar with `DataFrame.koalas.transform_batch()`; however, it takes a pandas Series as a chunk of Koalas Series.

```
>>> kdf = ks.DataFrame({'a': [1,2,3], 'b': [4,5,6]})
>>> def pandas_plus(pser):
...     return pser + 1 # should always return the same length as input.
...
>>> kdf.a.koalas.transform_batch(pandas_plus)
```

Under the hood, each batch of Koalas Series is split to multiple pandas Series, and each function computes on that as below:



There are more details such as the type inference and preventing its performance penalty. Please refer the API documentations.

## 2.4 Type Support In Koalas

In this chapter, we will briefly show you how data types change when converting Koalas DataFrame from/to PySpark DataFrame or pandas DataFrame.

### 2.4.1 Type casting between PySpark and Koalas

When converting a Koalas DataFrame from/to PySpark DataFrame, the data types are automatically casted to the appropriate type.

The example below shows how data types are casted from PySpark DataFrame to Koalas DataFrame.

```
# 1. Create a PySpark DataFrame
>>> sdf = spark.createDataFrame([
...     (1, Decimal(1.0), 1., 1., 1, 1, datetime(2020, 10, 27), "1", True,
...     datetime(2020, 10, 27)),
... ], 'tinyint tinyint, decimal decimal, float float, double double, integer integer,
...     long long, short short, timestamp timestamp, string string, boolean boolean, date,
...     date')

# 2. Check the PySpark data types
>>> sdf
```

(continues on next page)

(continued from previous page)

```
DataFrame[tinyint: tinyint, decimal: decimal(10,0), float: float, double: double,
↳ integer: int, long: bigint, short: smallint, timestamp: timestamp, string: string,
↳ boolean: boolean, date: date]

# 3. Convert PySpark DataFrame to Koalas DataFrame
>>> kdf = sdf.to_koalas()

# 4. Check the Koalas data types
>>> kdf.dtypes
tinyint          int8
decimal          object
float            float32
double           float64
integer          int32
long             int64
short            int16
timestamp        datetime64[ns]
string           object
boolean          bool
date             object
dtype: object
```

The example below shows how data types are casted from Koalas DataFrame to PySpark DataFrame.

```
# 1. Create a Koalas DataFrame
>>> kdf = ks.DataFrame({"int8": [1], "bool": [True], "float32": [1.0], "float64": [1.
↳ 0], "int32": [1], "int64": [1], "int16": [1], "datetime": [datetime.datetime(2020,
↳ 10, 27)], "object_string": ["1"], "object_decimal": [decimal.Decimal("1.1")],
↳ "object_date": [datetime.date(2020, 10, 27)]})

# 2. Type casting by using `astype`
>>> kdf['int8'] = kdf['int8'].astype('int8')
>>> kdf['int16'] = kdf['int16'].astype('int16')
>>> kdf['int32'] = kdf['int32'].astype('int32')
>>> kdf['float32'] = kdf['float32'].astype('float32')

# 3. Check the Koalas data types
>>> kdf.dtypes
int8          int8
bool          bool
float32       float32
float64       float64
int32         int32
int64         int64
int16         int16
datetime      datetime64[ns]
object_string object
object_decimal object
object_date   object
dtype: object

# 4. Convert Koalas DataFrame to PySpark DataFrame
>>> sdf = kdf.to_spark()

# 5. Check the PySpark data types
>>> sdf
DataFrame[int8: tinyint, bool: boolean, float32: float, float64: double, int32: int,
↳ int64: bigint, int16: smallint, datetime: timestamp, object_string: string, object_
↳ decimal: decimal(2,1), object_date: date]
```

(continues on next page)

(continued from previous page)

## 2.4.2 Type casting between pandas and Koalas

When converting Koalas DataFrame to pandas DataFrame, and the data types are basically same as pandas.

```
# Convert Koalas DataFrame to pandas DataFrame
>>> pdf = kdf.to_pandas()

# Check the pandas data types
>>> pdf.dtypes
int8          int8
bool          bool
float32       float32
float64       float64
int32         int32
int64         int64
int16         int16
datetime      datetime64[ns]
object_string object
object_decimal object
object_date   object
dtype: object
```

However, there are several data types only provided by pandas.

```
# pd.Catrgorical type is not supported in Koalas yet.
>>> ks.Series([pd.Categorical([1, 2, 3])])
Traceback (most recent call last):
...
pyarrow.lib.ArrowInvalid: Could not convert [1, 2, 3]
Categories (3, int64): [1, 2, 3] with type Categorical: did not recognize Python_
↪value type when inferring an Arrow data type
```

These kind of pandas specific data types below are not currently supported in Koalas but planned to be supported.

- `pd.Timedelta`
- `pd.Categorical`
- `pd.CategoricalDtype`

The pandas specific data types below are not planned to be supported in Koalas yet.

- `pd.SparseDtype`
- `pd.DatetimeTZDtype`
- `pd.UInt*Dtype`
- `pd.BooleanDtype`
- `pd.StringDtype`

### 2.4.3 Internal type mapping

The table below shows which NumPy data types are matched to which PySpark data types internally in Koalas.

NumPy	PySpark
np.character	BinaryType
np.bytes_	BinaryType
np.string_	BinaryType
np.int8	ByteType
np.byte	ByteType
np.int16	ShortType
np.int32	IntegerType
np.int64	LongType
np.int	LongType
np.float32	FloatType
np.float	DoubleType
np.float64	DoubleType
np.str	StringType
np.unicode_	StringType
np.bool	BooleanType
np.datetime64	TimestampType
np.ndarray	ArrayType(StringType())

The table below shows which Python data types are matched to which PySpark data types internally in Koalas.

Python	PySpark
bytes	BinaryType
int	LongType
float	DoubleType
str	StringType
bool	BooleanType
datetime.datetime	TimestampType
datetime.date	DateType
decimal.Decimal	DecimalType(38, 18)

For decimal type, Koalas uses Spark's system default precision and scale.

You can check this mapping by using `as_spark_type` function.

```
>>> import typing
>>> import numpy as np
>>> from databricks.koalas.typedef import as_spark_type

>>> as_spark_type(int)
LongType

>>> as_spark_type(np.int32)
IntegerType

>>> as_spark_type(typing.List[float])
ArrayType(DoubleType, true)
```

You can also check the underlying PySpark data type of *Series* or schema of *DataFrame* by using Spark accessor.

```
>>> ks.Series([0.3, 0.1, 0.8]).spark.data_type
DoubleType

>>> ks.Series(["welcome", "to", "Koalas"]).spark.data_type
StringType

>>> ks.Series([[False, True, False]]).spark.data_type
ArrayType(BooleanType,true)

>>> ks.DataFrame({"d": [0.3, 0.1, 0.8], "s": ["welcome", "to", "Koalas"], "b": [False,
↪ True, False]}).spark.print_schema()
root
 |-- d: double (nullable = false)
 |-- s: string (nullable = false)
 |-- b: boolean (nullable = false)
```

**Note:** Koalas currently does not support multiple types of data in single column.

```
>>> ks.Series([1, "A"])
Traceback (most recent call last):
...
TypeError: an integer is required (got type str)
```

## 2.5 Type Hints In Koalas

Koalas, by default, infers the schema by taking some top records from the output, in particular, when you use APIs that allow users to apply a function against Koalas DataFrame such as `DataFrame.transform()`, `DataFrame.apply()`, `DataFrame.koalas.apply_batch()`, `DataFrame.koalas.apply_batch()`, `Series.koalas.apply_batch()`, etc.

However, this is potentially expensive. If there are several expensive operations such as a shuffle in the upstream of the execution plan, Koalas will end up with executing the Spark job twice, once for schema inference, and once for processing actual data with the schema.

To avoid the consequences, Koalas has its own type hinting style to specify the schema to avoid schema inference. Koalas understands the type hints specified in the return type and converts it as a Spark schema for pandas UDFs used internally. The way of type hinting has been evolved over the time.

In this chapter, it covers the recommended way and the supported ways in details.

**Note:** The variadic generics support is experimental and unstable in Koalas. The way of typing can change between minor releases without a warning. See also [PEP 646](#) for variadic generics in Python.

## 2.5.1 Koalas DataFrame and Pandas DataFrame

In the early Koalas version, it was introduced to specify a type hint in the function in order to use it as a Spark schema. As an example, you can specify the return type hint as below by using Koalas `DataFrame`.

```
>>> def pandas_div(pdf) -> ks.DataFrame[float, float]:
...     # pdf is a pandas DataFrame.
...     return pdf[['B', 'C']] / pdf[['B', 'C']]
...
>>> df = ks.DataFrame({'A': ['a', 'a', 'b'], 'B': [1, 2, 3], 'C': [4, 6, 5]})
>>> df.groupby('A').apply(pandas_div)
```

The function `pandas_div` actually takes and outputs a pandas `DataFrame` instead of Koalas `DataFrame`. However, Koalas has to force to set the mismatched type hints.

From Koalas 1.0 with Python 3.7+, now you can specify the type hints by using pandas instances.

```
>>> def pandas_div(pdf) -> pd.DataFrame[float, float]:
...     # pdf is a pandas DataFrame.
...     return pdf[['B', 'C']] / pdf[['B', 'C']]
...
>>> df = ks.DataFrame({'A': ['a', 'a', 'b'], 'B': [1, 2, 3], 'C': [4, 6, 5]})
>>> df.groupby('A').apply(pandas_div)
```

Likewise, pandas Series can be also used as a type hints:

```
>>> def sqrt(x) -> pd.Series[float]:
...     return np.sqrt(x)
...
>>> df = ks.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
>>> df.apply(sqrt, axis=0)
```

Currently, both Koalas and pandas instances can be used to specify the type hints; however, Koalas plans to move gradually towards using pandas instances only as the stability becomes proven.

## 2.5.2 Type Hinting with Names

In Koalas 1.0, the new style of type hinting was introduced to overcome the limitations in the existing type hinting especially for `DataFrame`. When you use a `DataFrame` as the return type hint, for example, `DataFrame[int, int]`, there is no way to specify the names of each Series. In the old way, Koalas just generates the column names as `c#` and this easily leads users to lose or forget the Series mappings. See the example below:

```
>>> def transform(pdf) -> pd.DataFrame[int, int]:
...     pdf['A'] = pdf.id + 1
...     return pdf
...
>>> ks.range(5).koalas.apply_batch(transform)
```

```
   c0  c1
0    0   1
1    1   2
2    2   3
3    3   4
4    4   5
```



The new style of type hinting in Koalas is similar with the regular Python type hints in variables. The Series name is specified as a string, and the type is specified after a colon. The following example shows a simple case with the Series names, `id` and `A`, and `int` types respectively.

```
>>> def transform(pdf) -> pd.DataFrame["id": int, "A": int]:
...     pdf['A'] = pdf.id + 1
...     return pdf
...
>>> ks.range(5).koalas.apply_batch(transform)
```

```
   id  A
0    0  1
1    1  2
2    2  3
3    3  4
4    4  5
```

In addition, Koalas also dynamically supports `dtype` instance and the column index in pandas so that users can programmatically generate the return type and schema.

```
>>> def transform(pdf) -> pd.DataFrame[zip(pdf.columns, pdf.dtypes)]:
...     return pdf + 1
...
>>> kdf.koalas.apply_batch(transform)
```

Likewise, `dtype` instances from pandas `DataFrame` can be used alone and let Koalas generate column names.

```
>>> def transform(pdf) -> pd.DataFrame[pdf.dtypes]:
...     return pdf + 1
...
>>> kdf.koalas.apply_batch(transform)
```

## 2.6 From/to other DBMSes

The APIs interacting with other DBMSes in Koalas are slightly different from the ones in pandas because Koalas leverages JDBC APIs in PySpark to read and write from/to other DBMSes.

The APIs to read/write from/to external DBMSes are as follows:

<code>read_sql_table(table_name, con[, schema, ...])</code>	Read SQL database table into a <code>DataFrame</code> .
<code>read_sql_query(sql, con[, index_col])</code>	Read SQL query into a <code>DataFrame</code> .
<code>read_sql(sql, con[, index_col, columns])</code>	Read SQL query or database table into a <code>DataFrame</code> .

Koalas needs a canonical JDBC URL for `con`, and is able to take extra keyword arguments for the options in PySpark JDBC APIs:

```
ks.read_sql(..., dbtable="...", driver="", keytab="", ...)
```

## 2.6.1 Reading and writing DataFrames

In the example below, you will read and write a table in SQLite.

Firstly, create the example database as below via Python's SQLite library. This will be read to Koalas later:

```
import sqlite3

con = sqlite3.connect('example.db')
cur = con.cursor()
# Create table
cur.execute(
    '''CREATE TABLE stocks
       (date text, trans text, symbol text, qty real, price real)'''
)
# Insert a row of data
cur.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")
# Save (commit) the changes
con.commit()
con.close()
```

Koalas requires a JDBC driver to read so it requires the driver for your particular database to be on the Spark's classpath. For SQLite JDBC driver, you can download it, for example, as below:

```
curl -O https://repo1.maven.org/maven2/org/xerial/sqlite-jdbc/3.34.0/sqlite-jdbc-3.34.0.jar
```

After that, you should add it into your Spark session first. Once you add them, Koalas will automatically detect the Spark session and leverage it.

```
import os

from pyspark.sql import SparkSession

(SparkSession.builder
 .master("local")
 .appName("SQLite JDBC")
 .config(
     "spark.jars",
     "{}/sqlite-jdbc-3.34.0.jar".format(os.getcwd())
 )
 .config(
     "spark.driver.extraClassPath",
     "{}/sqlite-jdbc-3.34.0.jar".format(os.getcwd())
 )
 .getOrCreate())
```

Now, you're ready to read the table:

```
import databricks.koalas as ks

df = ks.read_sql("stocks", con="jdbc:sqlite:{}".format(os.getcwd()))
df
```

	date	trans	symbol	qty	price
0	2006-01-05	BUY	RHAT	100.0	35.14

You can also write it back to the `stocks` table as below:

```
df.price += 1
df.to_spark_io(
    format="jdbc", mode="append",
    dbtable="stocks", url="jdbc:sqlite:{}".format(os.getcwd()))
ks.read_sql("stocks", con="jdbc:sqlite:{}".format(os.getcwd()))
```

	date	trans	symbol	qty	price
0	2006-01-05	BUY	RHAT	100.0	35.14
1	2006-01-05	BUY	RHAT	100.0	36.14

## 2.7 Best Practices

### 2.7.1 Leverage PySpark APIs

Koalas uses Spark under the hood; therefore, many features and performance optimization are available in Koalas as well. Leverage and combine those cutting-edge features with Koalas.

Existing Spark context and Spark sessions are used out of the box in Koalas. If you already have your own configured Spark context or sessions running, Koalas uses them.

If there is no Spark context or session running in your environment (e.g., ordinary Python interpreter), such configurations can be set to `SparkContext` and/or `SparkSession`. Once Spark context and/or session is created, Koalas can use this context and/or session automatically. For example, if you want to configure the executor memory in Spark, you can do as below:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf()
conf.set('spark.executor.memory', '2g')
# Koalas automatically uses this Spark context with the configurations set.
SparkContext(conf=conf)

import databricks.koalas as ks
...
```

Another common configuration might be Arrow optimization in PySpark. In case of SQL configuration, it can be set into Spark session as below:

```
from pyspark.sql import SparkSession
builder = SparkSession.builder.appName("Koalas")
builder = builder.config("spark.sql.execution.arrow.enabled", "true")
# Koalas automatically uses this Spark session with the configurations set.
builder.getOrCreate()

import databricks.koalas as ks
...
```

All Spark features such as history server, web UI and deployment modes can be used as are with Koalas. If you are interested in performance tuning, please see also [Tuning Spark](#).

## 2.7.2 Check execution plans

Expensive operations can be predicted by leveraging PySpark API `DataFrame.spark.explain()` before the actual computation since Koalas is based on lazy execution. For example, see below.

```
>>> import databricks.koalas as ks
>>> kdf = ks.DataFrame({'id': range(10)})
>>> kdf = kdf[kdf.id > 5]
>>> kdf.spark.explain()
== Physical Plan ==
*(1) Filter (id#1L > 5)
+- *(1) Scan ExistingRDD[__index_level_0__#0L,id#1L]
```

Whenever you are not sure about such cases, you can check the actual execution plans and foresee the expensive cases.

Even though Koalas tries its best to optimize and reduce such shuffle operations by leveraging Spark optimizers, it is best to avoid shuffling in the application side whenever possible.

## 2.7.3 Use checkpoint

After a bunch of operations on Koalas objects, the underlying Spark planner can slow down due to the huge and complex plan. If the Spark plan becomes huge or it takes the planning long time, `DataFrame.spark.checkpoint()` or `DataFrame.spark.local_checkpoint()` would be helpful.

```
>>> import databricks.koalas as ks
>>> kdf = ks.DataFrame({'id': range(10)})
>>> kdf = kdf[kdf.id > 5]
>>> kdf['id'] = kdf['id'] + (10 * kdf['id'] + kdf['id'])
>>> kdf = kdf.groupby('id').head(2)
>>> kdf.spark.explain()
== Physical Plan ==
*(3) Project [__index_level_0__#0L, id#31L]
+- *(3) Filter (isnotnull(__row_number__#44) AND (__row_number__#44 <= 2))
    +- Window [row_number() window specification (__groupkey_0__#36L, __natural_order__
    ↪ #16L ASC NULLS FIRST, specified window frame (RowFrame, unbounded preceding $(), ↪
    ↪ current row $())) AS __row_number__#44], [__groupkey_0__#36L], [__natural_order__#16L,
    ↪ ASC NULLS FIRST]
        +- *(2) Sort [__groupkey_0__#36L ASC NULLS FIRST, __natural_order__#16L ASC,
    ↪ NULLS FIRST], false, 0
            +- Exchange hashpartitioning(__groupkey_0__#36L, 200), true, [id=#33]
                +- *(1) Project [__index_level_0__#0L, (id#1L + ((id#1L * 10) + id#1L)) ↪
    ↪ AS __groupkey_0__#36L, (id#1L + ((id#1L * 10) + id#1L)) AS id#31L, __natural_order__
    ↪ #16L]
                    +- *(1) Project [__index_level_0__#0L, id#1L, monotonically_increasing_
    ↪ id() AS __natural_order__#16L]
                        +- *(1) Filter (id#1L > 5)
                            +- *(1) Scan ExistingRDD[__index_level_0__#0L,id#1L]

>>> kdf = kdf.spark.local_checkpoint() # or kdf.spark.checkpoint()
>>> kdf.spark.explain()
== Physical Plan ==
*(1) Project [__index_level_0__#0L, id#31L]
+- *(1) Scan ExistingRDD[__index_level_0__#0L,id#31L,__natural_order__#59L]
```

As you can see, the previous Spark plan is dropped and starts with a simple plan. The result of the previous DataFrame is stored in the configured file system when calling `DataFrame.spark.checkpoint()`, or in the executor when calling `DataFrame.spark.local_checkpoint()`.

## 2.7.4 Avoid shuffling

Some operations such as `sort_values` are more difficult to do in a parallel or distributed environment than in in-memory on a single machine because it needs to send data to other nodes, and exchange the data across multiple nodes via networks. See the example below.

```
>>> import databricks.koalas as ks
>>> kdf = ks.DataFrame({'id': range(10)}).sort_values(by="id")
>>> kdf.spark.explain()
== Physical Plan ==
*(2) Sort [id#9L ASC NULLS LAST], true, 0
+- Exchange rangepartitioning(id#9L ASC NULLS LAST, 200), true, [id=#18]
   +- *(1) Scan ExistingRDD[__index_level_0__#8L,id#9L]
```

As you can see, it requires Exchange which requires a shuffle and it is likely expensive.

## 2.7.5 Avoid computation on single partition

Another common case is the computation on a single partition. Currently, some APIs such as `DataFrame.rank` uses PySpark's Window without specifying partition specification. This leads to move all data into a single partition in single machine and could cause serious performance degradation. Such APIs should be avoided very large dataset.

```
>>> import databricks.koalas as ks
>>> kdf = ks.DataFrame({'id': range(10)})
>>> kdf.rank().spark.explain()
== Physical Plan ==
*(4) Project [__index_level_0__#16L, id#24]
+- Window [avg(cast(_w0#26 as bigint)) windowSpecDefinition(id#17L,
↪specifiedwindowframe(RowFrame, unboundedpreceding$(), unboundedfollowing$())) AS id
↪#24], [id#17L]
   +- *(3) Project [__index_level_0__#16L, _w0#26, id#17L]
      +- Window [row_number() windowSpecDefinition(id#17L ASC NULLS FIRST,
↪specifiedwindowframe(RowFrame, unboundedpreceding$(), currentrow$())) AS _w0#26],
↪[id#17L ASC NULLS FIRST]
         +- *(2) Sort [id#17L ASC NULLS FIRST], false, 0
            +- Exchange SinglePartition, true, [id=#48]
               +- *(1) Scan ExistingRDD[__index_level_0__#16L,id#17L]
```

Instead, use `GroupBy.rank` as it is less expensive because data can be distributed and computed for each group.

## 2.7.6 Avoid reserved column names

Columns with leading `__` and trailing `__` are reserved in Koalas. To handle internal behaviors for, such as, index, Koalas uses some internal columns. Therefore, it is discouraged to use such column names and not guaranteed to work.

### 2.7.7 Do not use duplicated column names

It is disallowed to use duplicated column names because Spark SQL does not allow this in general. Koalas inherits this behavior. For instance, see below:

```
>>> import databricks.koalas as ks
>>> kdf = ks.DataFrame({'a': [1, 2], 'b': [3, 4]})
>>> kdf.columns = ["a", "a"]
...
Reference 'a' is ambiguous, could be: a, a.;
```

Additionally, it is strongly discouraged to use case sensitive column names. Koalas disallows it by default.

```
>>> import databricks.koalas as ks
>>> kdf = ks.DataFrame({'a': [1, 2], 'A': [3, 4]})
...
Reference 'a' is ambiguous, could be: a, A.;
```

However, you can turn on `spark.sql.caseSensitive` in Spark configuration to enable it if you use on your own risk.

```
>>> from pyspark.sql import SparkSession
>>> builder = SparkSession.builder.appName("Koalas")
>>> builder = builder.config("spark.sql.caseSensitive", "true")
>>> builder.getOrCreate()

>>> import databricks.koalas as ks
>>> kdf = ks.DataFrame({'a': [1, 2], 'A': [3, 4]})
>>> kdf
   a  A
0  1  3
1  2  4
```

### 2.7.8 Specify the index column in conversion from Spark DataFrame to Koalas DataFrame

When Koalas DataFrame is converted from Spark DataFrame, it loses the index information, which results in using the default index in Koalas DataFrame. The default index is inefficient in general comparing to explicitly specifying the index column. Specify the index column whenever possible.

See [working with PySpark](#)

### 2.7.9 Use distributed or distributed-sequence default index

One common issue when Koalas users face is the slow performance by default index. Koalas attaches a default index when the index is unknown, for example, Spark DataFrame is directly converted to Koalas DataFrame.

This default index is `sequence` which requires the computation on single partition which is discouraged. If you plan to handle large data in production, make it distributed by configuring the default index to `distributed` or `distributed-sequence`.

See [Default Index Type](#) for more details about configuring default index.

### 2.7.10 Reduce the operations on different DataFrame/Series

Koalas disallows the operations on different DataFrames (or Series) by default to prevent expensive operations. It internally performs a join operation which can be expensive in general, which is discouraged. Whenever possible, this operation should be avoided.

See *Operations on different DataFrames* for more details.

### 2.7.11 Use Koalas APIs directly whenever possible

Although Koalas has most of the pandas-equivalent APIs, there are several APIs not implemented yet or explicitly unsupported.

As an example, Koalas does not implement `__iter__()` to prevent users from collecting all data into the client (driver) side from the whole cluster. Unfortunately, many external APIs such as Python built-in functions such as `min`, `max`, `sum`, etc. require the given argument to be iterable. In case of pandas, it works properly out of the box as below:

```
>>> import pandas as pd
>>> max(pd.Series([1, 2, 3]))
3
>>> min(pd.Series([1, 2, 3]))
1
>>> sum(pd.Series([1, 2, 3]))
6
```

pandas dataset lives in the single machine, and is naturally iterable locally within the same machine. However, Koalas dataset lives across multiple machines, and they are computed in a distributed manner. It is difficult to be locally iterable and it is very likely users collect the entire data into the client side without knowing it. Therefore, it is best to stick to using Koalas APIs. The examples above can be converted as below:

```
>>> import databricks.koalas as ks
>>> ks.Series([1, 2, 3]).max()
3
>>> ks.Series([1, 2, 3]).min()
1
>>> ks.Series([1, 2, 3]).sum()
6
```

Another common pattern from pandas users might be to rely on list comprehension or generator expression. However, it also assumes the dataset is locally iterable under the hood. Therefore, it works seamlessly in pandas as below:

```
>>> import pandas as pd
>>> data = []
>>> countries = ['London', 'New York', 'Helsinki']
>>> pser = pd.Series([20., 21., 12.], index=countries)
>>> for temperature in pser:
...     assert temperature > 0
...     if temperature > 1000:
...         temperature = None
...     data.append(temperature ** 2)
...
>>> pd.Series(data, index=countries)
London      400.0
New York    441.0
Helsinki    144.0
dtype: float64
```

However, for Koalas it does not work as the same reason above. The example above can be also changed to directly using Koalas APIs as below:

```
>>> import databricks.koalas as ks
>>> import numpy as np
>>> countries = ['London', 'New York', 'Helsinki']
>>> kser = ks.Series([20., 21., 12.], index=countries)
>>> def square(temperature) -> np.float64:
...     assert temperature > 0
...     if temperature > 1000:
...         temperature = None
...     return temperature ** 2
...
>>> kser.apply(square)
London      400.0
New York    441.0
Helsinki    144.0
dtype: float64
```

## 2.8 FAQ

### 2.8.1 What's the project's status?

Koalas 1.0.0 was released, and it is much more stable now. You might still face the following differences:

- Most of pandas-equivalent APIs are implemented but still some may be missing. Please create a GitHub issue if your favorite function is not yet supported. We also document all APIs that are not yet supported in the [missing directory](#).
- Some behaviors may be different, in particular in the treatment of nulls: Pandas uses Not a Number (NaN) special constants to indicate missing values, while Spark has a special flag on each value to indicate missing values. We would love to hear from you if you come across any discrepancies
- Because Spark is lazy in nature, some operations like creating new columns only get performed when Spark needs to print or write the dataframe.

### 2.8.2 Is it Koalas or koalas?

It's Koalas. Unlike pandas, we use upper case here.

### 2.8.3 Should I use PySpark's DataFrame API or Koalas?

If you are already familiar with pandas and want to leverage Spark for big data, we recommend using Koalas. If you are learning Spark from ground up, we recommend you start with PySpark's API.



## 2.8.4 Does Koalas support Structured Streaming?

No, Koalas does not support Structured Streaming officially.

As a workaround, you can use Koalas APIs with *foreachBatch* in Structured Streaming which allows batch APIs:

```
>>> def func(batch_df, batch_id):
...     koalas_df = ks.DataFrame(batch_df)
...     koalas_df['a'] = 1
...     print(koalas_df)

>>> spark.readStream.format("rate").load().writeStream.foreachBatch(func).start()
      timestamp  value  a
0 2020-02-21 09:49:37.574    4  1
      timestamp  value  a
0 2020-02-21 09:49:38.574    5  1
...

```

## 2.8.5 How can I request support for a method?

File a GitHub issue: <https://github.com/databricks/koalas/issues>

Databricks customers are also welcome to file a support ticket to request a new feature.

## 2.8.6 How is Koalas different from Dask?

Different projects have different focuses. Spark is already deployed in virtually every organization, and often is the primary interface to the massive amount of data stored in data lakes. Koalas was inspired by Dask, and aims to make the transition from pandas to Spark easy for data scientists.

## 2.8.7 How can I contribute to Koalas?

See [Contributing Guide](#).

## 2.8.8 Why a new project (instead of putting this in Apache Spark itself)?

Two reasons:

1. We want a venue in which we can rapidly iterate and make new releases. The overhead of making a release as a separate project is minuscule (in the order of minutes). A release on Spark takes a lot longer (in the order of days)
2. Koalas takes a different approach that might contradict Spark's API design principles, and those principles cannot be changed lightly given the large user base of Spark. A new, separate project provides an opportunity for us to experiment with new design principles.



## API REFERENCE

### 3.1 Input/Output

#### 3.1.1 Data Generator

---

<code>range(start[, end, step, num_partitions])</code>	Create a DataFrame with some range of numbers.
--	--

---

##### **databricks.koalas.range**

`databricks.koalas.range` (*start: int, end: Optional[int] = None, step: int = 1, num\_partitions: Optional[int] = None*) → `databricks.koalas.frame.DataFrame`

Create a DataFrame with some range of numbers.

The resulting DataFrame has a single int64 column named `id`, containing elements in a range from `start` to `end` (exclusive) with step value `step`. If only the first parameter (i.e. `start`) is specified, we treat it as the end value with the start value being 0.

This is similar to the `range` function in `SparkSession` and is used primarily for testing.

##### **Parameters**

**start** [int] the start value (inclusive)

**end** [int, optional] the end value (exclusive)

**step** [int, optional, default 1] the incremental step

**num\_partitions** [int, optional] the number of partitions of the DataFrame

##### **Returns**

**DataFrame**

##### **Examples**

When the first parameter is specified, we generate a range of values up till that number.

```
>>> ks.range(5)
  id
0   0
1   1
2   2
```

(continues on next page)

(continued from previous page)

```
3  3
4  4
```

When start, end, and step are specified:

```
>>> ks.range(start = 100, end = 200, step = 20)
      id
0  100
1  120
2  140
3  160
4  180
```

### 3.1.2 Spark Metastore Table

<code>read_table(name[, index_col])</code>	Read a Spark table and return a DataFrame.
<code>DataFrame.to_table(name[, format, mode, ...])</code>	Write the DataFrame into a Spark table.

#### `databricks.koalas.read_table`

`databricks.koalas.read_table(name: str, index_col: Union[str, List[str], None] = None) → databricks.koalas.frame.DataFrame`  
Read a Spark table and return a DataFrame.

##### Parameters

**name** [string] Table name in Spark.

**index\_col** [str or list of str, optional, default: None] Index column of table in Spark.

##### Returns

**DataFrame**

See also:

`DataFrame.to_table`

`read_delta`

`read_parquet`

`read_spark_io`

##### Examples

```
>>> ks.range(1).to_table('%s.my_table' % db)
>>> ks.read_table('%s.my_table' % db)
      id
0     0
```

```
>>> ks.range(1).to_table('%s.my_table' % db, index_col="index")
>>> ks.read_table('%s.my_table' % db, index_col="index")
      id
```

(continues on next page)

(continued from previous page)

```
index
0      0
```

**databricks.koalas.DataFrame.to\_table**

`DataFrame.to_table` (*name*: str, *format*: Optional[str] = None, *mode*: str = 'overwrite', *partition\_cols*: Union[str, List[str], None] = None, *index\_col*: Union[str, List[str], None] = None, *\*\*options*) → None

Write the DataFrame into a Spark table. `DataFrame.spark.to_table()` is an alias of `DataFrame.to_table()`.

**Parameters**

**name** [str, required] Table name in Spark.

**format** [string, optional] Specifies the output data source format. Some common ones are:

- 'delta'
- 'parquet'
- 'orc'
- 'json'
- 'csv'

**mode** [str { 'append', 'overwrite', 'ignore', 'error', 'errorifexists' }, default] 'overwrite'. Specifies the behavior of the save operation when the table exists already.

- 'append': Append the new data to existing data.
- 'overwrite': Overwrite existing data.
- 'ignore': Silently ignore this operation if data already exists.
- 'error' or 'errorifexists': Throw an exception if data already exists.

**partition\_cols** [str or list of str, optional, default None] Names of partitioning columns

**index\_col**: str or list of str, optional, default: None Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**options** Additional options passed directly to Spark.

**Returns**

None

See also:

[`read\_table`](#)

[`DataFrame.to\_spark\_io`](#)

[`DataFrame.spark.to\_spark\_io`](#)

[`DataFrame.to\_parquet`](#)

## Examples

```
>>> df = ks.DataFrame(dict(
...     date=list(pd.date_range('2012-1-1 12:00:00', periods=3, freq='M')),
...     country=['KR', 'US', 'JP'],
...     code=[1, 2, 3]), columns=['date', 'country', 'code'])
>>> df
```

	date	country	code
0	2012-01-31 12:00:00	KR	1
1	2012-02-29 12:00:00	US	2
2	2012-03-31 12:00:00	JP	3

```
>>> df.to_table('%s.my_table' % db, partition_cols='date')
```

### 3.1.3 Delta Lake

<code>read_delta(path[, version, timestamp, index_col])</code>	Read a Delta Lake table on some file system and return a DataFrame.
<code>DataFrame.to_delta(path[, mode, ...])</code>	Write the DataFrame out as a Delta Lake table.

#### databricks.koalas.read\_delta

`databricks.koalas.read_delta` (*path*: str, *version*: Optional[str] = None, *timestamp*: Optional[str] = None, *index\_col*: Union[str, List[str], None] = None, *\*\*options*)  
 → databricks.koalas.frame.DataFrame

Read a Delta Lake table on some file system and return a DataFrame.

If the Delta Lake table is already stored in the catalog (aka the metastore), use 'read\_table'.

#### Parameters

**path** [string] Path to the Delta Lake table.

**version** [string, optional] Specifies the table version (based on Delta's internal transaction version) to read from, using Delta's time travel feature. This sets Delta's 'versionAsOf' option.

**timestamp** [string, optional] Specifies the table version (based on timestamp) to read from, using Delta's time travel feature. This must be a valid date or timestamp string in Spark, and sets Delta's 'timestampAsOf' option.

**index\_col** [str or list of str, optional, default: None] Index column of table in Spark.

**options** Additional options that can be passed onto Delta.

#### Returns

**DataFrame**

See also:

`DataFrame.to_delta`

`read_table`

`read_spark_io`

`read_parquet`

## Examples

```
>>> ks.range(1).to_delta('%s/read_delta/foo' % path)
>>> ks.read_delta('%s/read_delta/foo' % path)
   id
0    0
```

```
>>> ks.range(10, 15, num_partitions=1).to_delta('%s/read_delta/foo' % path, mode=
↳ 'overwrite')
>>> ks.read_delta('%s/read_delta/foo' % path)
   id
0   10
1   11
2   12
3   13
4   14
```

```
>>> ks.read_delta('%s/read_delta/foo' % path, version=0)
   id
0    0
```

You can preserve the index in the roundtrip as below.

```
>>> ks.range(10, 15, num_partitions=1).to_delta(
...     '%s/read_delta/bar' % path, index_col="index")
>>> ks.read_delta('%s/read_delta/bar' % path, index_col="index")
...
   id
index
0    10
1    11
2    12
3    13
4    14
```

## databricks.koalas.DataFrame.to\_delta

`DataFrame.to_delta(path: str, mode: str = 'overwrite', partition_cols: Union[str, List[str], None] = None, index_col: Union[str, List[str], None] = None, **options) → None`

Write the DataFrame out as a Delta Lake table.

### Parameters

**path** [str, required] Path to write to.

**mode** [str { 'append', 'overwrite', 'ignore', 'error', 'errorifexists' }, default] 'overwrite'. Specifies the behavior of the save operation when the destination exists already.

- 'append': Append the new data to existing data.
- 'overwrite': Overwrite existing data.
- 'ignore': Silently ignore this operation if data already exists.
- 'error' or 'errorifexists': Throw an exception if data already exists.

**partition\_cols** [str or list of str, optional, default None] Names of partitioning columns

**index\_col:** str or list of str, optional, default: None Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**options** [dict] All other options passed directly into Delta Lake.

See also:

`read_delta`

`DataFrame.to_parquet`

`DataFrame.to_table`

`DataFrame.to_spark_io`

## Examples

```
>>> df = ks.DataFrame(dict(
...     date=list(pd.date_range('2012-1-1 12:00:00', periods=3, freq='M')),
...     country=['KR', 'US', 'JP'],
...     code=[1, 2, 3]), columns=['date', 'country', 'code'])
>>> df
           date country  code
0 2012-01-31 12:00:00    KR    1
1 2012-02-29 12:00:00    US    2
2 2012-03-31 12:00:00    JP    3
```

Create a new Delta Lake table, partitioned by one column:

```
>>> df.to_delta('%s/to_delta/foo' % path, partition_cols='date')
```

Partitioned by two columns:

```
>>> df.to_delta('%s/to_delta/bar' % path, partition_cols=['date', 'country'])
```

Overwrite an existing table's partitions, using the 'replaceWhere' capability in Delta:

```
>>> df.to_delta('%s/to_delta/bar' % path,
...             mode='overwrite', replaceWhere='date >= "2012-01-01"')
```

### 3.1.4 Parquet

<code>read_parquet(path[, columns, index_col, ...])</code>	Load a parquet object from the file path, returning a DataFrame.
<code>DataFrame.to_parquet(path[, mode, ...])</code>	Write the DataFrame out as a Parquet file or directory.



**databricks.koalas.read\_parquet**

```
databricks.koalas.read_parquet(path, columns=None, index_col=None,
                               pandas_metadata=False, **options) →
                               databricks.koalas.frame.DataFrame
```

Load a parquet object from the file path, returning a DataFrame.

**Parameters**

**path** [string] File path

**columns** [list, default=None] If not None, only these columns will be read from the file.

**index\_col** [str or list of str, optional, default: None] Index column of table in Spark.

**pandas\_metadata** [bool, default: False] If True, try to respect the metadata if the Parquet file is written from pandas.

**options** [dict] All other options passed directly into Spark's data source.

**Returns**

**DataFrame**

See also:

[`DataFrame.to\_parquet`](#)

[`DataFrame.read\_table`](#)

[`DataFrame.read\_delta`](#)

[`DataFrame.read\_spark\_io`](#)

**Examples**

```
>>> ks.range(1).to_parquet('%s/read_spark_io/data.parquet' % path)
>>> ks.read_parquet('%s/read_spark_io/data.parquet' % path, columns=['id'])
   id
0    0
```

You can preserve the index in the roundtrip as below.

```
>>> ks.range(1).to_parquet('%s/read_spark_io/data.parquet' % path, index_col=
↳ "index")
>>> ks.read_parquet('%s/read_spark_io/data.parquet' % path, columns=['id'], index_
↳ col="index")
...
   id
index
0    0
```

**databricks.koalas.DataFrame.to\_parquet**

`DataFrame.to_parquet` (*path*: str, *mode*: str = 'overwrite', *partition\_cols*: Union[str, List[str], None] = None, *compression*: Optional[str] = None, *index\_col*: Union[str, List[str], None] = None, *\*\*options*) → None

Write the DataFrame out as a Parquet file or directory.

**Parameters**

**path** [str, required] Path to write to.

**mode** [str { 'append', 'overwrite', 'ignore', 'error', 'errorifexists' },] default 'overwrite'. Specifies the behavior of the save operation when the destination exists already.

- 'append': Append the new data to existing data.
- 'overwrite': Overwrite existing data.
- 'ignore': Silently ignore this operation if data already exists.
- 'error' or 'errorifexists': Throw an exception if data already exists.

**partition\_cols** [str or list of str, optional, default None] Names of partitioning columns

**compression** [str { 'none', 'uncompressed', 'snappy', 'gzip', 'lzo', 'brotli', 'lz4', 'zstd' }] Compression codec to use when saving to file. If None is set, it uses the value specified in `spark.sql.parquet.compression.codec`.

**index\_col**: str or list of str, optional, default: None Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**options** [dict] All other options passed directly into Spark's data source.

See also:

[`read\_parquet`](#)

[`DataFrame.to\_delta`](#)

[`DataFrame.to\_table`](#)

[`DataFrame.to\_spark\_io`](#)

**Examples**

```
>>> df = ks.DataFrame(dict(
...     date=list(pd.date_range('2012-1-1 12:00:00', periods=3, freq='M')),
...     country=['KR', 'US', 'JP'],
...     code=[1, 2, 3]), columns=['date', 'country', 'code'])
>>> df
```

	date	country	code
0	2012-01-31 12:00:00	KR	1
1	2012-02-29 12:00:00	US	2
2	2012-03-31 12:00:00	JP	3

```
>>> df.to_parquet('%s/to_parquet/foo.parquet' % path, partition_cols='date')
```

```
>>> df.to_parquet(
...     '%s/to_parquet/foo.parquet' % path,
...     mode = 'overwrite',
...     partition_cols=['date', 'country'])
```

### 3.1.5 ORC

<code>read_orc(path[, columns, index_col])</code>	Load an ORC object from the file path, returning a DataFrame.
<code>DataFrame.to_orc(path[, mode, ...])</code>	Write the DataFrame out as a ORC file or directory.

#### `databricks.koalas.read_orc`

`databricks.koalas.read_orc` (*path*, *columns*: *Optional[List[str]]* = *None*, *index\_col*: *Union[str, List[str], None]* = *None*, *\*\*options*) → `databricks.koalas.frame.DataFrame`  
 Load an ORC object from the file path, returning a DataFrame.

##### Parameters

**path** [str] The path string storing the ORC file to be read.

**columns** [list, default None] If not None, only these columns will be read from the file.

**index\_col** [str or list of str, optional, default: None] Index column of table in Spark.

**options** [dict] All other options passed directly into Spark's data source.

##### Returns

**DataFrame**

#### Examples

```
>>> ks.range(1).to_orc('%s/read_spark_io/data.orc' % path)
>>> ks.read_orc('%s/read_spark_io/data.orc' % path, columns=['id'])
   id
0   0
```

You can preserve the index in the roundtrip as below.

```
>>> ks.range(1).to_orc('%s/read_spark_io/data.orc' % path, index_col="index")
>>> ks.read_orc('%s/read_spark_io/data.orc' % path, columns=['id'], index_col=
↳ "index")
...
   id
index
0   0
```

**databricks.koalas.DataFrame.to\_orc**

`DataFrame.to_orc` (*path*: str, *mode*: str = 'overwrite', *partition\_cols*: Union[str, List[str], None] = None, *index\_col*: Union[str, List[str], None] = None, *\*\*options*) → None

Write the DataFrame out as a ORC file or directory.

**Parameters**

**path** [str, required] Path to write to.

**mode** [str {'append', 'overwrite', 'ignore', 'error', 'errorifexists'},] default 'overwrite'. Specifies the behavior of the save operation when the destination exists already.

- 'append': Append the new data to existing data.
- 'overwrite': Overwrite existing data.
- 'ignore': Silently ignore this operation if data already exists.
- 'error' or 'errorifexists': Throw an exception if data already exists.

**partition\_cols** [str or list of str, optional, default None] Names of partitioning columns

**index\_col**: str or list of str, optional, default: None Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**options** [dict] All other options passed directly into Spark's data source.

See also:

[`read\_orc`](#)

[`DataFrame.to\_delta`](#)

[`DataFrame.to\_parquet`](#)

[`DataFrame.to\_table`](#)

[`DataFrame.to\_spark\_io`](#)

**Examples**

```
>>> df = ks.DataFrame(dict(
...     date=list(pd.date_range('2012-1-1 12:00:00', periods=3, freq='M')),
...     country=['KR', 'US', 'JP'],
...     code=[1, 2, 3]), columns=['date', 'country', 'code'])
>>> df
           date country  code
0 2012-01-31 12:00:00    KR    1
1 2012-02-29 12:00:00    US    2
2 2012-03-31 12:00:00    JP    3
```

```
>>> df.to_orc('%s/to_orc/foo.orc' % path, partition_cols='date')
```

```
>>> df.to_orc(
...     '%s/to_orc/foo.orc' % path,
...     mode = 'overwrite',
...     partition_cols=['date', 'country'])
```

### 3.1.6 Generic Spark I/O

---

<code>read_spark_io([path, format, schema, index_col])</code>	Load a DataFrame from a Spark data source.
<code>DataFrame.to_spark_io([path, format, mode, ...])</code>	Write the DataFrame out to a Spark data source.

---

#### `databricks.koalas.read_spark_io`

`databricks.koalas.read_spark_io` (*path*: *Optional[str] = None*, *format*: *Optional[str] = None*, *schema*: *Union[str, StructType] = None*, *index\_col*: *Union[str, List[str], None] = None*, *\*\*options*)  
 → `databricks.koalas.frame.DataFrame`  
 Load a DataFrame from a Spark data source.

##### Parameters

**path** [string, optional] Path to the data source.

**format** [string, optional] Specifies the output data source format. Some common ones are:

- 'delta'
- 'parquet'
- 'orc'
- 'json'
- 'csv'

**schema** [string or StructType, optional] Input schema. If none, Spark tries to infer the schema automatically. The schema can either be a Spark StructType, or a DDL-formatted string like *col0 INT, col1 DOUBLE*.

**index\_col** [str or list of str, optional, default: None] Index column of table in Spark.

**options** [dict] All other options passed directly into Spark's data source.

See also:

`DataFrame.to_spark_io`

`DataFrame.read_table`

`DataFrame.read_delta`

`DataFrame.read_parquet`

##### Examples

```
>>> ks.range(1).to_spark_io('%s/read_spark_io/data.parquet' % path)
>>> ks.read_spark_io(
...     '%s/read_spark_io/data.parquet' % path, format='parquet', schema='id long
↳ ')
   id
0   0
```

```
>>> ks.range(10, 15, num_partitions=1).to_spark_io('%s/read_spark_io/data.json' %
↳ path,
...                                     format='json', lineSep='__')
>>> ks.read_spark_io(
...     '%s/read_spark_io/data.json' % path, format='json', schema='id long',
↳ lineSep='__')
   id
0  10
1  11
2  12
3  13
4  14
```

You can preserve the index in the roundtrip as below.

```
>>> ks.range(10, 15, num_partitions=1).to_spark_io('%s/read_spark_io/data.orc' %
↳ path,
...                                     format='orc', index_col="index"
↳ ")
>>> ks.read_spark_io(
...     path=r'%s/read_spark_io/data.orc' % path, format="orc", index_col="index")
...
   id
index
0    10
1    11
2    12
3    13
4    14
```

### `databricks.koalas.DataFrame.to_spark_io`

`DataFrame.to_spark_io` (*path*: *Optional[str]* = *None*, *format*: *Optional[str]* = *None*, *mode*: *str* = *'overwrite'*, *partition\_cols*: *Union[str, List[str], None]* = *None*, *index\_col*: *Union[str, List[str], None]* = *None*, *\*\*options*) → *None*

Write the `DataFrame` out to a Spark data source. `DataFrame.spark.to_spark_io()` is an alias of `DataFrame.to_spark_io()`.

#### Parameters

**path** [string, optional] Path to the data source.

**format** [string, optional] Specifies the output data source format. Some common ones are:

- 'delta'
- 'parquet'
- 'orc'
- 'json'
- 'csv'

**mode** [str { 'append', 'overwrite', 'ignore', 'error', 'errorifexists' }, default] 'overwrite'. Specifies the behavior of the save operation when data already.

- 'append': Append the new data to existing data.
- 'overwrite': Overwrite existing data.

- 'ignore': Silently ignore this operation if data already exists.
- 'error' or 'errorifexists': Throw an exception if data already exists.

**partition\_cols** [str or list of str, optional] Names of partitioning columns

**index\_col**: str or list of str, optional, default: None Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**options** [dict] All other options passed directly into Spark's data source.

#### Returns

None

See also:

`read_spark_io`

`DataFrame.to_delta`

`DataFrame.to_parquet`

`DataFrame.to_table`

`DataFrame.to_spark_io`

`DataFrame.spark.to_spark_io`

#### Examples

```
>>> df = ks.DataFrame(dict(
...     date=list(pd.date_range('2012-1-1 12:00:00', periods=3, freq='M')),
...     country=['KR', 'US', 'JP'],
...     code=[1, 2, 3]), columns=['date', 'country', 'code'])
>>> df
```

	date	country	code
0	2012-01-31 12:00:00	KR	1
1	2012-02-29 12:00:00	US	2
2	2012-03-31 12:00:00	JP	3

```
>>> df.to_spark_io(path='%s/to_spark_io/foo.json' % path, format='json')
```

### 3.1.7 Flat File / CSV

<code>read_csv(path[, sep, header, names, ...])</code>	Read CSV (comma-separated) file into DataFrame or Series.
<code>DataFrame.to_csv([path, sep, na_rep, ...])</code>	Write object to a comma-separated values (csv) file.

**databricks.koalas.read\_csv**

```
databricks.koalas.read_csv(path, sep=',', header='infer', names=None, index_col=None, usecols=None, squeeze=False, mangle_dupe_cols=True, dtype=None, nrows=None, parse_dates=False, quotechar=None, escapechar=None, comment=None, **options) → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]
```

Read CSV (comma-separated) file into DataFrame or Series.

**Parameters**

**path** [str] The path string storing the CSV file to be read.

**sep** [str, default ','] Delimiter to use. Must be a single character.

**header** [int, list of int, default 'infer'] Whether to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to *header=0* and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to *header=None*. Explicitly pass *header=0* to be able to replace existing names

**names** [str or array-like, optional] List of column names to use. If file contains no header row, then you should explicitly pass *header=None*. Duplicates in this list will cause an error to be issued. If a string is given, it should be a DDL-formatted string in Spark SQL, which is preferred to avoid schema inference for better performance.

**index\_col: str or list of str, optional, default: None** Index column of table in Spark.

**usecols** [list-like or callable, optional] Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in names or inferred from the document header row(s). If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to *True*.

**squeeze** [bool, default False] If the parsed data only contains one column then return a Series.

**mangle\_dupe\_cols** [bool, default True] Duplicate columns will be specified as 'X0', 'X1', ... 'XN', rather than 'X' ... 'X'. Passing in False will cause data to be overwritten if there are duplicate names in the columns. Currently only *True* is allowed.

**dtype** [Type name or dict of column -> type, default None] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} Use str or object together with suitable *na\_values* settings to preserve and not interpret dtype.

**nrows** [int, default None] Number of rows to read from the CSV file.

**parse\_dates** [boolean or list of ints or names or list of lists or dict, default *False*.] Currently only *False* is allowed.

**quotechar** [str (length 1), optional] The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

**escapechar** [str (length 1), default None] One-character string used to escape delimiter

**comment: str, optional** Indicates the line should not be parsed.

**options** [dict] All other options passed directly into Spark's data source.

**Returns**

**DataFrame or Series**

See also:



**DataFrame.to\_csv** Write DataFrame to a comma-separated values (csv) file.

## Examples

```
>>> ks.read_csv('data.csv')
```

## databricks.koalas.DataFrame.to\_csv

**DataFrame.to\_csv** (*path=None, sep=','*, *na\_rep=""*, *columns=None*, *header=True*, *quotechar=""*, *date\_format=None*, *escapechar=None*, *num\_files=None*, *mode: str = 'overwrite'*, *partition\_cols: Union[str, List[str], None] = None*, *index\_col: Union[str, List[str], None] = None*, *\*\*options*) → Optional[str]

Write object to a comma-separated values (csv) file.

---

**Note:** Koalas *to\_csv* writes files to a path or URI. Unlike pandas', Koalas respects HDFS's property such as 'fs.default.name'.

---



---

**Note:** Koalas writes CSV files into the directory, *path*, and writes multiple *part-...* files in the directory when *path* is specified. This behaviour was inherited from Apache Spark. The number of files can be controlled by *num\_files*.

---

## Parameters

- path** [str, default None] File path. If None is provided the result is returned as a string.
- sep** [str, default ','] String of length 1. Field delimiter for the output file.
- na\_rep** [str, default ''] Missing data representation.
- columns** [sequence, optional] Columns to write.
- header** [bool or list of str, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.
- quotechar** [str, default '"'] String of length 1. Character used to quote fields.
- date\_format** [str, default None] Format string for datetime objects.
- escapechar** [str, default None] String of length 1. Character used to escape *sep* and *quotechar* when appropriate.
- num\_files** [the number of files to be written in *path* directory when] this is a path.
- mode** [str { 'append', 'overwrite', 'ignore', 'error', 'errorifexists' },] default 'overwrite'. Specifies the behavior of the save operation when the destination exists already.
  - 'append': Append the new data to existing data.
  - 'overwrite': Overwrite existing data.
  - 'ignore': Silently ignore this operation if data already exists.
  - 'error' or 'errorifexists': Throw an exception if data already exists.
- partition\_cols** [str or list of str, optional, default None] Names of partitioning columns

**index\_col:** str or list of str, optional, default: None Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**options:** keyword arguments for additional options specific to PySpark. This kwarg is specific to PySpark's CSV options to pass. Check the options in PySpark's API documentation for `spark.write.csv(...)`. It has higher priority and overwrites all other options. This parameter only works when `path` is specified.

### Returns

str or None

See also:

[`read\_csv`](#)

[`DataFrame.to\_delta`](#)

[`DataFrame.to\_table`](#)

[`DataFrame.to\_parquet`](#)

[`DataFrame.to\_spark\_io`](#)

### Examples

```
>>> df = ks.DataFrame(dict(
...     date=list(pd.date_range('2012-1-1 12:00:00', periods=3, freq='M')),
...     country=['KR', 'US', 'JP'],
...     code=[1, 2, 3]), columns=['date', 'country', 'code'])
>>> df.sort_values(by="date")
           date country  code
... 2012-01-31 12:00:00    KR    1
... 2012-02-29 12:00:00    US    2
... 2012-03-31 12:00:00    JP    3
```

```
>>> print(df.to_csv())
date,country,code
2012-01-31 12:00:00,KR,1
2012-02-29 12:00:00,US,2
2012-03-31 12:00:00,JP,3
```

```
>>> df.cummax().to_csv(path=r'%s/to_csv/foo.csv' % path, num_files=1)
>>> ks.read_csv(
...     path=r'%s/to_csv/foo.csv' % path
... ).sort_values(by="date")
           date country  code
... 2012-01-31 12:00:00    KR    1
... 2012-02-29 12:00:00    US    2
... 2012-03-31 12:00:00    US    3
```

In case of Series,

```
>>> print(df.date.to_csv())
date
2012-01-31 12:00:00
```

(continues on next page)

(continued from previous page)

```
2012-02-29 12:00:00
2012-03-31 12:00:00
```

```
>>> df.date.to_csv(path=r'%s/to_csv/foo.csv' % path, num_files=1)
>>> ks.read_csv(
...     path=r'%s/to_csv/foo.csv' % path
... ).sort_values(by="date")
...     date
... 2012-01-31 12:00:00
... 2012-02-29 12:00:00
... 2012-03-31 12:00:00
```

You can preserve the index in the roundtrip as below.

```
>>> df.set_index("country", append=True, inplace=True)
>>> df.date.to_csv(
...     path=r'%s/to_csv/bar.csv' % path,
...     num_files=1,
...     index_col=["index1", "index2"])
>>> ks.read_csv(
...     path=r'%s/to_csv/bar.csv' % path, index_col=["index1", "index2"]
... ).sort_values(by="date")
...     date
index1 index2
...     ...   2012-01-31 12:00:00
...     ...   2012-02-29 12:00:00
...     ...   2012-03-31 12:00:00
```

### 3.1.8 Clipboard

<code>read_clipboard([sep])</code>	Read text from clipboard and pass to read_csv.
<code>DataFrame.to_clipboard([excel, sep])</code>	Copy object to the system clipboard.

#### databricks.koalas.read\_clipboard

`databricks.koalas.read_clipboard(sep='\s+', **kwargs) → databricks.koalas.frame.DataFrame`  
 Read text from clipboard and pass to read\_csv. See read\_csv for the full argument list

##### Parameters

**sep** [str, default 's+'] A string or regex delimiter. The default of 's+' denotes one or more whitespace characters.

##### Returns

**parsed** [DataFrame]

See also:

`DataFrame.to_clipboard` Write text out to clipboard.

## `databricks.koalas.DataFrame.to_clipboard`

`DataFrame.to_clipboard(excel=True, sep=None, **kwargs) → None`

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

---

**Note:** This method should only be used if the resulting DataFrame is expected to be small, as all the data is loaded into the driver's memory.

---

### Parameters

**excel** [bool, default True]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

**sep** [str, default '\t'] Field delimiter.

**\*\*kwargs** These parameters will be passed to `DataFrame.to_csv`.

See also:

[`read\_clipboard`](#) Read text from clipboard.

### Notes

Requirements for your platform.

- Linux : `xclip`, or `xsel` (with `gtk` or `PyQt4` modules)
- Windows : none
- OS X : none

### Examples

Copy the contents of a DataFrame to the clipboard.

```
>>> df = ks.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the index by passing the keyword `index` and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

This function also works for Series:

```

>>> df = ks.Series([1, 2, 3, 4, 5, 6, 7], name='x')
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # 0, 1
... # 1, 2
... # 2, 3
... # 3, 4
... # 4, 5
... # 5, 6
... # 6, 7

```

### 3.1.9 Excel

<code>read_excel(io[, sheet_name, header, names, ...])</code>	Read an Excel file into a Koalas DataFrame or Series.
<code>DataFrame.to_excel(excel_writer[, ...])</code>	Write object to an Excel sheet.

#### databricks.koalas.read\_excel

`databricks.koalas.read_excel` (*io*, *sheet\_name=0*, *header=0*, *names=None*, *index\_col=None*, *usecols=None*, *squeeze=False*, *dtype=None*, *engine=None*, *converters=None*, *true\_values=None*, *false\_values=None*, *skiprows=None*, *nrows=None*, *na\_values=None*, *keep\_default\_na=True*, *verbose=False*, *parse\_dates=False*, *date\_parser=None*, *thousands=None*, *comment=None*, *skipfooter=0*, *convert\_float=True*, *mangle\_dupe\_cols=True*, *\*\*kws*) → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series, collections.OrderedDict]

Read an Excel file into a Koalas DataFrame or Series.

Support both *xls* and *xlsx* file extensions from a local filesystem or URL. Support an option to read a single sheet or a list of sheets.

#### Parameters

**io** [str, file descriptor, pathlib.Path, ExcelFile or xlrd.Book] The string could be a URL. The value URL must be available in Spark's DataFrameReader.

---

**Note:** If the underlying Spark is below 3.0, the parameter as a string is not supported. You can use `ks.from_pandas(pd.read_excel(...))` as a workaround.

---

**sheet\_name** [str, int, list, or None, default 0] Strings are used for sheet names. Integers are used in zero-indexed sheet positions. Lists of strings/integers are used to request multiple sheets. Specify None to get all sheets.

Available cases:

- Defaults to 0: 1st sheet as a *DataFrame*
- 1: 2nd sheet as a *DataFrame*
- "Sheet1": Load sheet with name "Sheet1"
- [0, 1, "Sheet5"]: Load first, second and sheet named "Sheet5" as a dict of *DataFrame*

- None: All sheets.

**header** [int, list of int, default 0] Row (0-indexed) to use for the column labels of the parsed DataFrame. If a list of integers is passed those row positions will be combined into a `MultiIndex`. Use None if there is no header.

**names** [array-like, default None] List of column names to use. If file contains no header row, then you should explicitly pass `header=None`.

**index\_col** [int, list of int, default None] Column (0-indexed) to use as the row labels of the DataFrame. Pass None if there is no such column. If a list is passed, those columns will be combined into a `MultiIndex`. If a subset of data is selected with `usecols`, `index_col` is based on the subset.

**usecols** [int, str, list-like, or callable default None] Return a subset of the columns.

- If None, then parse all columns.
- If str, then indicates comma separated list of Excel column letters and column ranges (e.g. "A:E" or "A,C,E:F"). Ranges are inclusive of both sides.
- If list of int, then indicates list of column numbers to be parsed.
- If list of string, then indicates list of column names to be parsed.
- If callable, then evaluate each column name against it and parse the column if the callable returns `True`.

**squeeze** [bool, default False] If the parsed data only contains one column then return a Series.

**dtype** [Type name or dict of column -> type, default None] Data type for data or columns. E.g. `{ 'a': np.float64, 'b': np.int32 }` Use *object* to preserve data as stored in Excel and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

**engine** [str, default None] If io is not a buffer or path, this must be set to identify io. Acceptable values are None or `xlrd`.

**converters** [dict, default None] Dict of functions for converting values in certain columns. Keys can either be integers or column labels, values are functions that take one input argument, the Excel cell content, and return the transformed content.

**true\_values** [list, default None] Values to consider as True.

**false\_values** [list, default None] Values to consider as False.

**skiprows** [list-like] Rows to skip at the beginning (0-indexed).

**nrows** [int, default None] Number of rows to parse.

**na\_values** [scalar, str, list-like, or dict, default None] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN.

**keep\_default\_na** [bool, default True] If `na_values` are specified and `keep_default_na` is False the default NaN values are overridden, otherwise they're appended to.

**verbose** [bool, default False] Indicate number of NA values placed in non-numeric columns.

**parse\_dates** [bool, list-like, or dict, default False] The behavior is as follows:

- bool. If True -> try parsing the index.
- list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.

- list of lists. e.g. If `[[1, 3]]` -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. `{ 'foo' : [1, 3] }` -> parse columns 1, 3 as date and call result 'foo'

If a column or index contains an unparseable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `pd.to_datetime` after `pd.read_csv`

Note: A fast-path exists for iso8601-formatted dates.

**date\_parser** [function, optional] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Koalas will try to call `date_parser` in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by `parse_dates`) as arguments; 2) concatenate (row-wise) the string values from the columns defined by `parse_dates` into a single array and pass that; and 3) call `date_parser` once for each row using one or more strings (corresponding to the columns defined by `parse_dates`) as arguments.

**thousands** [str, default None] Thousands separator for parsing string columns to numeric. Note that this parameter is only necessary for columns stored as TEXT in Excel, any numeric columns will automatically be parsed, regardless of display format.

**comment** [str, default None] Comments out remainder of line. Pass a character or characters to this argument to indicate comments in the input file. Any data between the comment string and the end of the current line is ignored.

**skipfooter** [int, default 0] Rows at the end to skip (0-indexed).

**convert\_float** [bool, default True] Convert integral floats to int (i.e., 1.0 -> 1). If False, all numeric data will be read in as floats: Excel stores all numbers as floats internally.

**mangle\_dupe\_cols** [bool, default True] Duplicate columns will be specified as 'X', 'X.1', ... 'X.N', rather than 'X'...'X'. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

**\*\*kwargs** [optional] Optional keyword arguments can be passed to `TextFileReader`.

## Returns

**DataFrame or dict of DataFrames** DataFrame from the passed in Excel file. See notes in `sheet_name` argument for more information on when a dict of DataFrames is returned.

See also:

[`DataFrame.to\_excel`](#) Write DataFrame to an Excel file.

[`DataFrame.to\_csv`](#) Write DataFrame to a comma-separated values (csv) file.

[`read\_csv`](#) Read a comma-separated values (csv) file into DataFrame.

## Examples

The file can be read using the file name as string or an open file object:

```
>>> ks.read_excel('tmp.xlsx', index_col=0)
   Name  Value
0  string1     1
1  string2     2
2  #Comment     3
```

```
>>> ks.read_excel(open('tmp.xlsx', 'rb'),
...               sheet_name='Sheet3')
   Unnamed: 0    Name  Value
0           0    string1    1
1           1    string2    2
2           2  #Comment    3
```

Index and header can be specified via the *index\_col* and *header* arguments

```
>>> ks.read_excel('tmp.xlsx', index_col=None, header=None)
   0      1      2
0 NaN    Name  Value
1 0.0  string1    1
2 1.0  string2    2
3 2.0  #Comment    3
```

Column types are inferred but can be explicitly specified

```
>>> ks.read_excel('tmp.xlsx', index_col=0,
...               dtype={'Name': str, 'Value': float})
   Name  Value
0  string1  1.0
1  string2  2.0
2  #Comment 3.0
```

True, False, and NA values, and thousands separators have defaults, but can be explicitly specified, too. Supply the values you would like as strings or lists of strings!

```
>>> ks.read_excel('tmp.xlsx', index_col=0,
...               na_values=['string1', 'string2'])
   Name  Value
0   None    1
1   None    2
2  #Comment    3
```

Comment lines in the excel input file can be skipped using the *comment* kwarg

```
>>> ks.read_excel('tmp.xlsx', index_col=0, comment='#')
   Name  Value
0  string1  1.0
1  string2  2.0
2   None   NaN
```

### `databricks.koalas.DataFrame.to_excel`

```
DataFrame.to_excel(excel_writer, sheet_name='Sheet1', na_rep="", float_format=None,
                   columns=None, header=True, index=True, index_label=None, startrow=0,
                   startcol=0, engine=None, merge_cells=True, encoding=None, inf_rep='inf',
                   verbose=True, freeze_panes=None) → None
```

Write object to an Excel sheet.

**Note:** This method should only be used if the resulting DataFrame is expected to be small, as all the data is loaded into the driver's memory.



To write a single object to an Excel .xlsx file it is only necessary to specify a target file name. To write to multiple sheets it is necessary to create an *ExcelWriter* object with a target file name, and specify a sheet in the file to write to.

Multiple sheets may be written to by specifying unique *sheet\_name*. With all data written to the file it is necessary to save the changes. Note that creating an *ExcelWriter* object with a file name that already exists will result in the contents of the existing file being erased.

#### Parameters

- excel\_writer** [str or ExcelWriter object] File path or existing ExcelWriter.
- sheet\_name** [str, default 'Sheet1'] Name of sheet which will contain DataFrame.
- na\_rep** [str, default ''] Missing data representation.
- float\_format** [str, optional] Format string for floating point numbers. For example `float_format="%.2f"` will format 0.1234 to 0.12.
- columns** [sequence or list of str, optional] Columns to write.
- header** [bool or list of str, default True] Write out the column names. If a list of string is given it is assumed to be aliases for the column names.
- index** [bool, default True] Write row names (index).
- index\_label** [str or sequence, optional] Column label for index column(s) if desired. If not specified, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.
- startrow** [int, default 0] Upper left cell row to dump data frame.
- startcol** [int, default 0] Upper left cell column to dump data frame.
- engine** [str, optional] Write engine to use, 'openpyxl' or 'xlsxwriter'. You can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.
- merge\_cells** [bool, default True] Write MultiIndex and Hierarchical Rows as merged cells.
- encoding** [str, optional] Encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.
- inf\_rep** [str, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel).
- verbose** [bool, default True] Display more information in the error logs.
- freeze\_panes** [tuple of int (length 2), optional] Specifies the one-based bottommost row and rightmost column that is to be frozen.

See also:

[`read\_excel`](#) Read Excel file.

## Notes

Once a workbook has been saved it is not possible write further data without rewriting the whole workbook.

## Examples

Create, write to and save a workbook:

```
>>> df1 = ks.DataFrame([[ 'a', 'b'], [ 'c', 'd']],
...                     index=[ 'row 1', 'row 2'],
...                     columns=[ 'col 1', 'col 2'])
>>> df1.to_excel("output.xlsx")
```

To specify the sheet name:

```
>>> df1.to_excel("output.xlsx")
>>> df1.to_excel("output.xlsx",
...              sheet_name='Sheet_name_1')
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:

```
>>> with pd.ExcelWriter('output.xlsx') as writer:
...     df1.to_excel(writer, sheet_name='Sheet_name_1')
...     df2.to_excel(writer, sheet_name='Sheet_name_2')
```

To set the library that is used to write the Excel file, you can pass the *engine* keyword (the default engine is automatically chosen depending on the file extension):

```
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter')
```

### 3.1.10 JSON

<code>read_json(path[, lines, index_col])</code>	Convert a JSON string to DataFrame.
<code>DataFrame.to_json([path, compression, ...])</code>	Convert the object to a JSON string.

#### databricks.koalas.read\_json

`databricks.koalas.read_json` (*path*: str, *lines*: bool = True, *index\_col*: Union[str, List[str], None] = None, *\*\*options*) → databricks.koalas.frame.DataFrame

Convert a JSON string to DataFrame.

##### Parameters

**path** [string] File path

**lines** [bool, default True] Read the file as a json object per line. It should be always True for now.

**index\_col** [str or list of str, optional, default: None] Index column of table in Spark.

**options** [dict] All other options passed directly into Spark's data source.

## Examples

```
>>> df = ks.DataFrame([[ 'a', 'b'], [ 'c', 'd']],
...                     columns=[ 'col 1', 'col 2'])
```

```
>>> df.to_json(path=r'%s/read_json/foo.json' % path, num_files=1)
>>> ks.read_json(
...     path=r'%s/read_json/foo.json' % path
... ).sort_values(by="col 1")
   col 1 col 2
0      a      b
1      c      d
```

```
>>> df.to_json(path=r'%s/read_json/foo.json' % path, num_files=1, lineSep='___')
>>> ks.read_json(
...     path=r'%s/read_json/foo.json' % path, lineSep='___'
... ).sort_values(by="col 1")
   col 1 col 2
0      a      b
1      c      d
```

You can preserve the index in the roundtrip as below.

```
>>> df.to_json(path=r'%s/read_json/bar.json' % path, num_files=1, index_col="index
↪")
>>> ks.read_json(
...     path=r'%s/read_json/bar.json' % path, index_col="index"
... ).sort_values(by="col 1")
   col 1 col 2
index
0      a      b
1      c      d
```

## databricks.koalas.DataFrame.to\_json

`DataFrame.to_json` (*path=None*, *compression='uncompressed'*, *num\_files=None*, *mode: str = 'overwrite'*, *orient='records'*, *lines=True*, *partition\_cols: Union[str, List[str], None] = None*, *index\_col: Union[str, List[str], None] = None*, *\*\*options*) → `Optional[str]`

Convert the object to a JSON string.

---

**Note:** Koalas `to_json` writes files to a path or URI. Unlike pandas', Koalas respects HDFS's property such as `'fs.default.name'`.

---



---

**Note:** Koalas writes JSON files into the directory, *path*, and writes multiple *part-...* files in the directory when *path* is specified. This behaviour was inherited from Apache Spark. The number of files can be controlled by *num\_files*.

---



---

**Note:** output JSON format is different from pandas'. It always use *orient='records'* for its output. This behaviour might have to change in the near future.

---

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

### Parameters

**path** [string, optional] File path. If not specified, the result is returned as a string.

**lines** [bool, default True] If 'orient' is 'records' write out line delimited json format. Will throw ValueError if incorrect 'orient' since others are not list like. It should be always True for now.

**orient** [str, default 'records'] It should be always 'records' for now.

**compression** [{ 'gzip', 'bz2', 'xz', None}] A string representing the compression to use in the output file, only used when the first argument is a filename. By default, the compression is inferred from the filename.

**num\_files** [the number of files to be written in *path* directory when] this is a path.

**mode** [str { 'append', 'overwrite', 'ignore', 'error', 'errorifexists' },] default 'overwrite'. Specifies the behavior of the save operation when the destination exists already.

- 'append': Append the new data to existing data.
- 'overwrite': Overwrite existing data.
- 'ignore': Silently ignore this operation if data already exists.
- 'error' or 'errorifexists': Throw an exception if data already exists.

**partition\_cols** [str or list of str, optional, default None] Names of partitioning columns

**index\_col: str or list of str, optional, default: None** Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**options: keyword arguments for additional options specific to PySpark.** It is specific to PySpark's JSON options to pass. Check the options in PySpark's API documentation for *spark.write.json(...)*. It has a higher priority and overwrites all other options. This parameter only works when *path* is specified.

### Returns

**str or None**

### Examples

```
>>> df = ks.DataFrame([[ 'a', 'b'], [ 'c', 'd']],
...                    columns=[ 'col 1', 'col 2'])
>>> df.to_json()
' [{"col 1": "a", "col 2": "b"}, {"col 1": "c", "col 2": "d"} ] '
```

```
>>> df['col 1'].to_json()
' [{"col 1": "a"}, {"col 1": "c"} ] '
```

```
>>> df.to_json(path=r'%s/to_json/foo.json' % path, num_files=1)
>>> ks.read_json(
...     path=r'%s/to_json/foo.json' % path
... ).sort_values(by="col 1")
col 1 col 2
0     a     b
1     c     d
```

```
>>> df['col 1'].to_json(path=r'%s/to_json/foo.json' % path, num_files=1, index_
↳ col="index")
>>> ks.read_json(
...     path=r'%s/to_json/foo.json' % path, index_col="index"
... ).sort_values(by="col 1")
   col 1
index
0       a
1       c
```

### 3.1.11 HTML

<code>read_html(io[, match, flavor, header, ...])</code>	Read HTML tables into a list of DataFrame objects.
<code>DataFrame.to_html([buf, columns, col_space, ...])</code>	Render a DataFrame as an HTML table.

#### databricks.koalas.read\_html

`databricks.koalas.read_html` (*io*, *match*='+', *flavor*=None, *header*=None, *index\_col*=None, *skiprows*=None, *attrs*=None, *parse\_dates*=False, *thousands*=',', *encoding*=None, *decimal*='.', *converters*=None, *na\_values*=None, *keep\_default\_na*=True, *displayed\_only*=True) → List[databricks.koalas.frame.DataFrame]

Read HTML tables into a list of DataFrame objects.

#### Parameters

- io** [str or file-like] A URL, a file-like object, or a raw string containing HTML. Note that `lxml` only accepts the http, ftp and file url protocols. If you have a URL that starts with 'https' you might try removing the 's'.
- match** [str or compiled regular expression, optional] The set of tables containing text matching this regex or string will be returned. Unless the HTML is extremely simple you will probably need to pass a non-empty string here. Defaults to '+' (match any non-empty string). The default value will return all tables contained on a page. This value is converted to a regular expression so that there is consistent behavior between Beautiful Soup and `lxml`.
- flavor** [str or None, container of strings] The parsing engine to use. 'bs4' and 'html5lib' are synonymous with each other, they are both there for backwards compatibility. The default of None tries to use `lxml` to parse and if that fails it falls back on `bs4 + html5lib`.
- header** [int or list-like or None, optional] The row (or list of rows for a `MultiIndex`) to use to make the columns headers.
- index\_col** [int or list-like or None, optional] The column (or list of columns) to use to create the index.
- skiprows** [int or list-like or slice or None, optional] 0-based. Number of rows to skip after parsing the column integer. If a sequence of integers or a slice is given, will skip the rows indexed by that sequence. Note that a single element sequence means 'skip the nth row' whereas an integer means 'skip n rows'.
- attrs** [dict or None, optional] This is a dictionary of attributes that you can pass to use to identify the table in the HTML. These are not checked for validity before being passed to `lxml` or

Beautiful Soup. However, these attributes must be valid HTML table attributes to work correctly. For example,

```
attrs = {'id': 'table'}
```

is a valid attribute dictionary because the ‘id’ HTML tag attribute is a valid HTML attribute for *any* HTML tag as per [this document](#).

```
attrs = {'asdf': 'table'}
```

is *not* a valid attribute dictionary because ‘asdf’ is not a valid HTML attribute even if it is a valid XML attribute. Valid HTML 4.01 table attributes can be found [here](#). A working draft of the HTML 5 spec can be found [here](#). It contains the latest information on table attributes for the modern web.

**parse\_dates** [bool, optional] See `read_csv()` for more details.

**thousands** [str, optional] Separator to use to parse thousands. Defaults to ‘,’.

**encoding** [str or None, optional] The encoding used to decode the web page. Defaults to None. ``None`` preserves the previous encoding behavior, which depends on the underlying parser library (e.g., the parser library will try to use the encoding provided by the document).

**decimal** [str, default ‘.’] Character to recognize as decimal point (e.g. use ‘,’ for European data).

**converters** [dict, default None] Dict of functions for converting values in certain columns. Keys can either be integers or column labels, values are functions that take one input argument, the cell (not column) content, and return the transformed content.

**na\_values** [iterable, default None] Custom NA values

**keep\_default\_na** [bool, default True] If na\_values are specified and keep\_default\_na is False the default NaN values are overridden, otherwise they’re appended to

**displayed\_only** [bool, default True] Whether elements with “display: none” should be parsed

## Returns

**dfs** [list of DataFrames]

See also:

[`read\_csv`](#)

[`DataFrame.to\_html`](#)

## `databricks.koalas.DataFrame.to_html`

`DataFrame.to_html` (*buf=None, columns=None, col\_space=None, header=True, index=True, na\_rep='NaN', formatters=None, float\_format=None, sparsify=None, index\_names=True, justify=None, max\_rows=None, max\_cols=None, show\_dimensions=False, decimal='.', bold\_rows=True, classes=None, escape=True, notebook=False, border=None, table\_id=None, render\_links=False*)  
→ Optional[str]

Render a DataFrame as an HTML table.

---

**Note:** This method should only be used if the resulting pandas object is expected to be small, as all the data is loaded into the driver’s memory. If the input is large, set `max_rows` parameter.

---

**Parameters**

**buf** [StringIO-like, optional] Buffer to write to.

**columns** [sequence, optional, default None] The subset of columns to write. Writes all columns by default.

**col\_space** [int, optional] The minimum width of each column.

**header** [bool, optional] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names

**index** [bool, optional, default True] Whether to print index (row) labels.

**na\_rep** [str, optional, default 'NaN'] String representation of NAN to use.

**formatters** [list or dict of one-param. functions, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

**float\_format** [one-parameter function, optional, default None] Formatter function to apply to columns' elements if they are floats. The result of this function must be a unicode string.

**sparsify** [bool, optional, default True] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row.

**index\_names** [bool, optional, default True] Prints the names of the indexes.

**justify** [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by set\_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset.

**max\_rows** [int, optional] Maximum number of rows to display in the console.

**max\_cols** [int, optional] Maximum number of columns to display in the console.

**show\_dimensions** [bool, default False] Display DataFrame dimensions (number of rows by number of columns).

**decimal** [str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

**bold\_rows** [bool, default True] Make the row labels bold in the output.

**classes** [str or list or tuple, default None] CSS class(es) to apply to the resulting html table.

**escape** [bool, default True] Convert the characters <, >, and & to HTML-safe sequences.

**notebook** [{True, False}, default False] Whether the generated HTML is for IPython Notebook.

**border** [int] A border=border attribute is included in the opening `<table>` tag. Default `pd.options.html.border`.

**table\_id** [str, optional] A css id is included in the opening `<table>` tag if specified.

**render\_links** [bool, default False] Convert URLs to HTML links (only works with pandas 0.24+).

#### Returns

**str (or unicode, depending on data and options)** String representation of the dataframe.

#### See also:

[`to\_string`](#) Convert DataFrame to a string.

### 3.1.12 SQL

<code>read_sql_table(table_name, con[, schema, ...])</code>	Read SQL database table into a DataFrame.
<code>read_sql_query(sql, con[, index_col])</code>	Read SQL query into a DataFrame.
<code>read_sql(sql, con[, index_col, columns])</code>	Read SQL query or database table into a DataFrame.

#### databricks.koalas.read\_sql\_table

`databricks.koalas.read_sql_table` (*table\_name*, *con*, *schema=None*, *index\_col=None*, *columns=None*, *options=None*) → `databricks.koalas.frame.DataFrame`

Read SQL database table into a DataFrame.

Given a table name and a JDBC URI, returns a DataFrame.

#### Parameters

**table\_name** [str] Name of SQL table in database.

**con** [str] A JDBC URI could be provided as str.

---

**Note:** The URI must be JDBC URI instead of Python's database URI.

---

**schema** [str, default None] Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).

**index\_col** [str or list of str, optional, default: None] Column(s) to set as index(MultiIndex).

**columns** [list, default None] List of column names to select from SQL table.

**options** [dict] All other options passed directly into Spark's JDBC data source.

#### Returns

**DataFrame** A SQL table is returned as two-dimensional data structure with labeled axes.

#### See also:

[`read\_sql\_query`](#) Read SQL query into a DataFrame.

[`read\_sql`](#) Read SQL query or database table into a DataFrame.



## Examples

```
>>> ks.read_sql_table('table_name', 'jdbc:postgresql:db_name')
```

### databricks.koalas.read\_sql\_query

`databricks.koalas.read_sql_query` (*sql*, *con*, *index\_col=None*, *\*\*options*) → `databricks.koalas.frame.DataFrame`

Read SQL query into a DataFrame.

Returns a DataFrame corresponding to the result set of the query string. Optionally provide an *index\_col* parameter to use one of the columns as the index, otherwise default index will be used.

---

**Note:** Some database might hit the issue of Spark: SPARK-27596

---

#### Parameters

**sql** [string SQL query] SQL query to be executed.

**con** [str] A JDBC URI could be provided as str.

---

**Note:** The URI must be JDBC URI instead of Python's database URI.

---

**index\_col** [string or list of strings, optional, default: None] Column(s) to set as index(MultiIndex).

**options** [dict] All other options passed directly into Spark's JDBC data source.

#### Returns

**DataFrame**

See also:

[`read\_sql\_table`](#) Read SQL database table into a DataFrame.

[`read\_sql`](#)

## Examples

```
>>> ks.read_sql_query('SELECT * FROM table_name', 'jdbc:postgresql:db_name')
```

### databricks.koalas.read\_sql

`databricks.koalas.read_sql` (*sql*, *con*, *index\_col=None*, *columns=None*, *\*\*options*) → `databricks.koalas.frame.DataFrame`

Read SQL query or database table into a DataFrame.

This function is a convenience wrapper around `read_sql_table` and `read_sql_query` (for backward compatibility). It will delegate to the specific function depending on the provided input. A SQL query will be routed to `read_sql_query`, while a database table name will be routed to `read_sql_table`. Note that the delegated function might have more specific notes about their functionality not listed here.

---

**Note:** Some database might hit the issue of Spark: SPARK-27596

---

### Parameters

**sql** [string] SQL query to be executed or a table name.

**con** [str] A JDBC URI could be provided as as str.

---

**Note:** The URI must be JDBC URI instead of Python's database URI.

---

**index\_col** [string or list of strings, optional, default: None] Column(s) to set as index(MultiIndex).

**columns** [list, default: None] List of column names to select from SQL table (only used when reading a table).

**options** [dict] All other options passed directly into Spark's JDBC data source.

### Returns

**DataFrame**

See also:

[\*read\\_sql\\_table\*](#) Read SQL database table into a DataFrame.

[\*read\\_sql\\_query\*](#) Read SQL query into a DataFrame.

### Examples

```
>>> ks.read_sql('table_name', 'jdbc:postgresql:db_name')
>>> ks.read_sql('SELECT * FROM table_name', 'jdbc:postgresql:db_name')
```

## 3.2 General functions

### 3.2.1 Working with options

<code>reset_option(key)</code>	Reset one option to their default value.
<code>get_option(key[, default])</code>	Retrieves the value of the specified option.
<code>set_option(key, value)</code>	Sets the value of the specified option.
<code>option_context(*args)</code>	Context manager to temporarily set options in the <i>with</i> statement context.

**databricks.koalas.reset\_option**

`databricks.koalas.reset_option(key: str) → None`

Reset one option to their default value.

Pass “all” as argument to reset all options.

**Parameters**

**key** [str] If specified only option will be reset.

**Returns**

**None**

**databricks.koalas.get\_option**

`databricks.koalas.get_option(key: str, default: Union[Any, pyspark_globals.NoValueType] = <no value>) → Any`

Retrieves the value of the specified option.

**Parameters**

**key** [str] The key which should match a single option.

**default** [object] The default value if the option is not set yet. The value should be JSON serializable.

**Returns**

**result** [the value of the option]

**Raises**

**OptionError** [if no such option exists and the default is not provided]

**databricks.koalas.set\_option**

`databricks.koalas.set_option(key: str, value: Any) → None`

Sets the value of the specified option.

**Parameters**

**key** [str] The key which should match a single option.

**value** [object] New value of option. The value should be JSON serializable.

**Returns**

**None**

**databricks.koalas.option\_context**

`databricks.koalas.option_context(*args)`

Context manager to temporarily set options in the *with* statement context.

You need to invoke as `option_context(pat, val, [(pat, val), ...])`.

**Examples**

```
>>> with option_context('display.max_rows', 10, 'compute.max_rows', 5):
...     print(get_option('display.max_rows'), get_option('compute.max_rows'))
10 5
>>> print(get_option('display.max_rows'), get_option('compute.max_rows'))
1000 1000
```

**3.2.2 Data manipulations and SQL**

<code>melt(frame[, id_vars, value_vars, var_name, ...])</code>	Unpivot a DataFrame from wide format to long format, optionally leaving identifier variables set.
<code>merge(obj, right[, how, on, left_on, ...])</code>	Merge DataFrame objects with a database-style join.
<code>get_dummies(data[, prefix, prefix_sep, ...])</code>	Convert categorical variable into dummy/indicator variables, also known as one hot encoding.
<code>concat(objs[, axis, join, ignore_index, sort])</code>	Concatenate Koalas objects along a particular axis with optional set logic along the other axes.
<code>sql(query[, globals, locals])</code>	Execute a SQL query and return the result as a Koalas DataFrame.
<code>broadcast(obj)</code>	Marks a DataFrame as small enough for use in broadcast joins.

**databricks.koalas.melt**

`databricks.koalas.melt(frame, id_vars=None, value_vars=None, var_name=None, value_name='value') → databricks.koalas.frame.DataFrame`

Unpivot a DataFrame from wide format to long format, optionally leaving identifier variables set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id\_vars*), while all other columns, considered measured variables (*value\_vars*), are “unpivoted” to the row axis, leaving just two non-identifier columns, ‘variable’ and ‘value’.

**Parameters**

**frame** [DataFrame]

**id\_vars** [tuple, list, or ndarray, optional] Column(s) to use as identifier variables.

**value\_vars** [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id\_vars*.

**var\_name** [scalar, default ‘variable’] Name to use for the ‘variable’ column. If None it uses *frame.columns.name* or ‘variable’.

**value\_name** [scalar, default ‘value’] Name to use for the ‘value’ column.

**Returns**

**DataFrame** Unpivoted DataFrame.**Examples**

```
>>> df = ks.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}},
...                   columns=['A', 'B', 'C'])
>>> df
```

	A	B	C
0	a	1	2
1	b	3	4
2	c	5	6

```
>>> ks.melt(df)
variable value
0          A      a
1          B      1
2          C      2
3          A      b
4          B      3
5          C      4
6          A      c
7          B      5
8          C      6
```

```
>>> df.melt(id_vars='A')
A variable value
0 a          B      1
1 a          C      2
2 b          B      3
3 b          C      4
4 c          B      5
5 c          C      6
```

```
>>> df.melt(value_vars='A')
variable value
0          A      a
1          A      b
2          A      c
```

```
>>> ks.melt(df, id_vars=['A', 'B'])
A B variable value
0 a 1          C      2
1 b 3          C      4
2 c 5          C      6
```

```
>>> df.melt(id_vars=['A'], value_vars=['C'])
A variable value
0 a          C      2
1 b          C      4
2 c          C      6
```

The names of 'variable' and 'value' columns can be customized:

```
>>> ks.melt(df, id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
  A myVarname myValname
0  a          B         1
1  b          B         3
2  c          B         5
```

## databricks.koalas.merge

`databricks.koalas.merge` (*obj*, *right*: `databricks.koalas.frame.DataFrame`, *how*: *str* = 'inner', *on*: `Union[Any, List[Any], Tuple, List[Tuple]]` = *None*, *left\_on*: `Union[Any, List[Any], Tuple, List[Tuple]]` = *None*, *right\_on*: `Union[Any, List[Any], Tuple, List[Tuple]]` = *None*, *left\_index*: *bool* = *False*, *right\_index*: *bool* = *False*, *suffixes*: `Tuple[str, str]` = ('\_x', '\_y') → `databricks.koalas.frame.DataFrame`  
Merge DataFrame objects with a database-style join.

**The index of the resulting DataFrame will be one of the following:**

- 0...n if no index is used for merging
- Index of the left DataFrame if merged only on the index of the right DataFrame
- Index of the right DataFrame if merged only on the index of the left DataFrame
- **All involved indices if merged using the indices of both DataFrames** e.g. if *left* with indices (a, x) and *right* with indices (b, x), the result will be an index (x, a, b)

### Parameters

**right:** Object to merge with.

**how:** Type of merge to be performed. {'left', 'right', 'outer', 'inner'}, default 'inner'

**left:** use only keys from left frame, similar to a SQL left outer join; preserve key order.

**right:** use only keys from right frame, similar to a SQL right outer join; preserve key order.

**outer:** use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.

**inner:** use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.

**on:** Column or index level names to join on. These must be found in both DataFrames. If *on* is *None* and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

**left\_on:** Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

**right\_on:** Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

**left\_index:** Use the index from the left DataFrame as the join key(s). If it is a `MultiIndex`, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

**right\_index:** Use the index from the right DataFrame as the join key. Same caveats as left\_index.

**suffixes:** Suffix to apply to overlapping column names in the left and right side, respectively.

### Returns

**DataFrame** A DataFrame of the two merged objects.

### Notes

As described in #263, joining string columns currently returns None for missing values instead of NaN.

### Examples

```
>>> df1 = ks.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [1, 2, 3, 5]},
...                     columns=['lkey', 'value'])
>>> df2 = ks.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [5, 6, 7, 8]},
...                     columns=['rkey', 'value'])
>>> df1
   lkey  value
0  foo      1
1  bar      2
2  baz      3
3  foo      5
>>> df2
   rkey  value
0  foo      5
1  bar      6
2  baz      7
3  foo      8
```

Merge df1 and df2 on the lkey and rkey columns. The value columns have the default suffixes, \_x and \_y, appended.

```
>>> merged = ks.merge(df1, df2, left_on='lkey', right_on='rkey')
>>> merged.sort_values(by=['lkey', 'value_x', 'rkey', 'value_y'])
   lkey  value_x rkey  value_y
...bar      2  bar      6
...baz      3  baz      7
...foo      1  foo      5
...foo      1  foo      8
...foo      5  foo      5
...foo      5  foo      8
```

```
>>> left_kdf = ks.DataFrame({'A': [1, 2]})
>>> right_kdf = ks.DataFrame({'B': ['x', 'y']}, index=[1, 2])
```

```
>>> ks.merge(left_kdf, right_kdf, left_index=True, right_index=True).sort_index()
   A  B
1  2  x
```

```
>>> ks.merge(left_kdf, right_kdf, left_index=True, right_index=True, how='left').
↳ sort_index()
      A      B
0  1  None
1  2      x
```

```
>>> ks.merge(left_kdf, right_kdf, left_index=True, right_index=True, how='right').
↳ sort_index()
      A      B
1  2.0    x
2  NaN    y
```

```
>>> ks.merge(left_kdf, right_kdf, left_index=True, right_index=True, how='outer').
↳ sort_index()
      A      B
0  1.0  None
1  2.0     x
2  NaN     y
```

## databricks.koalas.get\_dummies

`databricks.koalas.get_dummies` (*data*, *prefix=None*, *prefix\_sep='\_'*, *dummy\_na=False*, *columns=None*, *sparse=False*, *drop\_first=False*, *dtype=None*)  
 → `databricks.koalas.frame.DataFrame`

Convert categorical variable into dummy/indicator variables, also known as one hot encoding.

### Parameters

**data** [array-like, Series, or DataFrame]

**prefix** [string, list of strings, or dict of strings, default None] String to append DataFrame column names. Pass a list with length equal to the number of columns when calling `get_dummies` on a DataFrame. Alternatively, *prefix* can be a dictionary mapping column names to prefixes.

**prefix\_sep** [string, default '\_'] If appending prefix, separator/delimiter to use. Or pass a list or dictionary as with *prefix*.

**dummy\_na** [bool, default False] Add a column to indicate NaNs, if False NaNs are ignored.

**columns** [list-like, default None] Column names in the DataFrame to be encoded. If *columns* is None then all the columns with *object* or *category* dtype will be converted.

**sparse** [bool, default False] Whether the dummy-encoded columns should be backed by a `SparseArray` (True) or a regular NumPy array (False). In Koalas, this value must be "False".

**drop\_first** [bool, default False] Whether to get k-1 dummies out of k categorical levels by removing the first level.

**dtype** [dtype, default np.uint8] Data type for new columns. Only a single dtype is allowed.

### Returns

**dummies** [DataFrame]

See also:

[`Series.str.get\_dummies`](#)



## Examples

```
>>> s = ks.Series(list('abca'))
```

```
>>> ks.get_dummies(s)
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
```

```
>>> df = ks.DataFrame({'A': ['a', 'b', 'a'], 'B': ['b', 'a', 'c'],
...                    'C': [1, 2, 3]},
...                   columns=['A', 'B', 'C'])
```

```
>>> ks.get_dummies(df, prefix=['col1', 'col2'])
   C  col1_a  col1_b  col2_a  col2_b  col2_c
0  1         1         0         0         1         0
1  2         0         1         1         0         0
2  3         1         0         0         0         1
```

```
>>> ks.get_dummies(ks.Series(list('abcaa')))
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
4  1  0  0
```

```
>>> ks.get_dummies(ks.Series(list('abcaa')), drop_first=True)
   b  c
0  0  0
1  1  0
2  0  1
3  0  0
4  0  0
```

```
>>> ks.get_dummies(ks.Series(list('abc')), dtype=float)
   a    b    c
0  1.0  0.0  0.0
1  0.0  1.0  0.0
2  0.0  0.0  1.0
```

## databricks.koalas.concat

`databricks.koalas.concat` (*objs*, *axis=0*, *join='outer'*, *ignore\_index=False*, *sort=False*) → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]

Concatenate Koalas objects along a particular axis with optional set logic along the other axes.

### Parameters

**objs** [a sequence of Series or DataFrame] Any None objects will be dropped silently unless they are all None in which case a ValueError will be raised

**axis** [{0/'index', 1/'columns'}, default 0] The axis to concatenate along.

**join** [{ 'inner', 'outer' }, default 'outer'] How to handle indexes on other axis (or axes).

**ignore\_index** [bool, default False] If True, do not use the index values along the concatenation axis. The resulting axis will be labeled 0, ..., n - 1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information. Note the index values on the other axes are still respected in the join.

**sort** [bool, default False] Sort non-concatenation axis if it is not already aligned.

### Returns

**object, type of objs** When concatenating all `Series` along the index (`axis=0`), a `Series` is returned. When `objs` contains at least one `DataFrame`, a `DataFrame` is returned. When concatenating along the columns (`axis=1`), a `DataFrame` is returned.

See also:

`Series.append` Concatenate Series.

`DataFrame.join` Join DataFrames using indexes.

`DataFrame.merge` Merge DataFrames by indexes or columns.

### Examples

```
>>> from databricks.koalas.config import set_option, reset_option
>>> set_option("compute.ops_on_diff_frames", True)
```

Combine two `Series`.

```
>>> s1 = ks.Series(['a', 'b'])
>>> s2 = ks.Series(['c', 'd'])
>>> ks.concat([s1, s2])
0    a
1    b
0    c
1    d
dtype: object
```

Clear the existing index and reset it in the result by setting the `ignore_index` option to `True`.

```
>>> ks.concat([s1, s2], ignore_index=True)
0    a
1    b
2    c
3    d
dtype: object
```

Combine two `DataFrame` objects with identical columns.

```
>>> df1 = ks.DataFrame([['a', 1], ['b', 2]],
...                     columns=['letter', 'number'])
>>> df1
  letter  number
0     a         1
1     b         2
>>> df2 = ks.DataFrame([['c', 3], ['d', 4]],
```

(continues on next page)

(continued from previous page)

```
... columns=['letter', 'number'])
>>> df2
  letter  number
0      c       3
1      d       4
```

```
>>> ks.concat([df1, df2])
  letter  number
0      a       1
1      b       2
0      c       3
1      d       4
```

Combine DataFrame and Series objects with different columns.

```
>>> ks.concat([df2, s1])
  letter  number  0
0      c       3.0  None
1      d       4.0  None
0  None     NaN    a
1  None     NaN    b
```

Combine DataFrame objects with overlapping columns and return everything. Columns outside the intersection will be filled with None values.

```
>>> df3 = ks.DataFrame([['c', 3, 'cat'], ['d', 4, 'dog']],
...                     columns=['letter', 'number', 'animal'])
>>> df3
  letter  number  animal
0      c       3    cat
1      d       4    dog
```

```
>>> ks.concat([df1, df3])
  letter  number  animal
0      a       1    None
1      b       2    None
0      c       3    cat
1      d       4    dog
```

Sort the columns.

```
>>> ks.concat([df1, df3], sort=True)
  animal  letter  number
0  None      a       1
1  None      b       2
0   cat      c       3
1   dog      d       4
```

Combine DataFrame objects with overlapping columns and return only those that are shared by passing inner to the join keyword argument.

```
>>> ks.concat([df1, df3], join="inner")
  letter  number
0      a       1
1      b       2
```

(continues on next page)

(continued from previous page)

0	c	3
1	d	4

```
>>> df4 = ks.DataFrame(['bird', 'polly'], ['monkey', 'george'],
...                      columns=['animal', 'name'])
```

Combine with column axis.

```
>>> ks.concat([df1, df4], axis=1)
  letter  number  animal  name
0      a        1    bird  polly
1      b        2  monkey  george
```

```
>>> reset_option("compute.ops_on_diff_frames")
```

## databricks.koalas.sql

`databricks.koalas.sql` (*query*: *str*, *globals*=None, *locals*=None, *\*\*kwargs*) → `databricks.koalas.frame.DataFrame`  
 Execute a SQL query and return the result as a Koalas DataFrame.

This function also supports embedding Python variables (locals, globals, and parameters) in the SQL statement by wrapping them in curly braces. See examples section for details.

In addition to the locals, globals and parameters, the function will also attempt to determine if the program currently runs in an IPython (or Jupyter) environment and to import the variables from this environment. The variables have the same precedence as globals.

The following variable types are supported:

- string
- int
- float
- list, tuple, range of above types
- Koalas DataFrame
- Koalas Series
- pandas DataFrame

### Parameters

**query** [str] the SQL query

**globals** [dict, optional] the dictionary of global variables, if explicitly set by the user

**locals** [dict, optional] the dictionary of local variables, if explicitly set by the user

**kwargs** other variables that the user may want to set manually that can be referenced in the query

### Returns

**Koalas DataFrame**

## Examples

Calling a built-in SQL function.

```
>>> ks.sql("select * from range(10) where id > 7")
   id
0    8
1    9
```

A query can also reference a local variable or parameter by wrapping them in curly braces:

```
>>> bound1 = 7
>>> ks.sql("select * from range(10) where id > {bound1} and id < {bound2}",
↳bound2=9)
   id
0    8
```

You can also wrap a DataFrame with curly braces to query it directly. Note that when you do that, the indexes, if any, automatically become top level columns.

```
>>> mydf = ks.range(10)
>>> x = range(4)
>>> ks.sql("SELECT * from {mydf} WHERE id IN {x}")
   id
0    0
1    1
2    2
3    3
```

Queries can also be arbitrarily nested in functions:

```
>>> def statement():
...     mydf2 = ks.DataFrame({"x": range(2)})
...     return ks.sql("SELECT * from {mydf2}")
>>> statement()
   x
0  0
1  1
```

Mixing Koalas and pandas DataFrames in a join operation. Note that the index is dropped.

```
>>> ks.sql('''
...     SELECT m1.a, m2.b
...     FROM {table1} m1 INNER JOIN {table2} m2
...     ON m1.key = m2.key
...     ORDER BY m1.a, m2.b''',
... table1=ks.DataFrame({"a": [1,2], "key": ["a", "b"]}),
... table2=pd.DataFrame({"b": [3,4,5], "key": ["a", "b", "b"]}))
   a  b
0  1  3
1  2  4
2  2  5
```

Also, it is possible to query using Series.

```
>>> myser = ks.Series({'a': [1.0, 2.0, 3.0], 'b': [15.0, 30.0, 45.0]})
>>> ks.sql("SELECT * from {myser}")
```

(continues on next page)

(continued from previous page)

```

      0
0      [1.0, 2.0, 3.0]
1      [15.0, 30.0, 45.0]

```

**databricks.koalas.broadcast**

`databricks.koalas.broadcast(obj)` → `databricks.koalas.frame.DataFrame`

Marks a DataFrame as small enough for use in broadcast joins.

**Parameters**

**obj** [DataFrame]

**Returns**

**ret** [DataFrame with broadcast hint.]

See also:

**DataFrame.merge** Merge DataFrame objects with a database-style join.

**DataFrame.join** Join columns of another DataFrame.

**DataFrame.update** Modify in place using non-NA values from another DataFrame.

**DataFrame.hint** Specifies some hint on the current DataFrame.

**Examples**

```

>>> df1 = ks.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [1, 2, 3, 5]},
...                     columns=['lkey', 'value']).set_index('lkey')
>>> df2 = ks.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [5, 6, 7, 8]},
...                     columns=['rkey', 'value']).set_index('rkey')
>>> merged = df1.merge(ks.broadcast(df2), left_index=True, right_index=True)
>>> merged.spark.explain()
== Physical Plan ==
...
...BroadcastHashJoin...
...

```

**3.2.3 Top-level missing data**

<code>to_numeric(arg)</code>	Convert argument to a numeric type.
<code>isna(obj)</code>	Detect missing values for an array-like object.
<code>isnull(obj)</code>	Detect missing values for an array-like object.
<code>notna(obj)</code>	Detect existing (non-missing) values.
<code>notnull(obj)</code>	Detect existing (non-missing) values.

**databricks.koalas.to\_numeric**`databricks.koalas.to_numeric(arg)`

Convert argument to a numeric type.

**Parameters****arg** [scalar, list, tuple, 1-d array, or Series]**Returns****ret** [numeric if parsing succeeded.]**See also:****DataFrame.astype** Cast argument to a specified dtype.**to\_datetime** Convert argument to datetime.**to\_timedelta** Convert argument to timedelta.**numpy.ndarray.astype** Cast a numpy array to a specified type.**Examples**

```
>>> kser = ks.Series(['1.0', '2', '-3'])
>>> kser
0    1.0
1     2
2    -3
dtype: object
```

```
>>> ks.to_numeric(kser)
0    1.0
1    2.0
2   -3.0
dtype: float32
```

If given Series contains invalid value to cast float, just cast it to *np.nan*

```
>>> kser = ks.Series(['apple', '1.0', '2', '-3'])
>>> kser
0    apple
1     1.0
2     2
3     -3
dtype: object
```

```
>>> ks.to_numeric(kser)
0    NaN
1    1.0
2    2.0
3   -3.0
dtype: float32
```

Also support for list, tuple, np.array, or a scalar

```
>>> ks.to_numeric(['1.0', '2', '-3'])
array([ 1.,  2., -3.])
```

```
>>> ks.to_numeric(['1.0', '2', '-3'])
array([ 1.,  2., -3.]
```

```
>>> ks.to_numeric(np.array(['1.0', '2', '-3']))
array([ 1.,  2., -3.]
```

```
>>> ks.to_numeric('1.0')
1.0
```

## databricks.koalas.isna

databricks.koalas.**isna** (*obj*)

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (NaN in numeric arrays, None or NaN in object arrays).

### Parameters

**obj** [scalar or array-like] Object to check for null or missing values.

### Returns

**bool or array-like of bool** For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is missing.

**See also:**

***Series.isna*** Detect missing values in a Series.

***Series.isnull*** Detect missing values in a Series.

***DataFrame.isna*** Detect missing values in a DataFrame.

***DataFrame.isnull*** Detect missing values in a DataFrame.

***Index.isna*** Detect missing values in an Index.

***Index.isnull*** Detect missing values in an Index.

## Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> ks.isna('dog')
False
```

```
>>> ks.isna(np.nan)
True
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> ks.isna(array)
```

(continues on next page)



(continued from previous page)

```
array([[False,  True, False],
       [False, False,  True]])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = ks.DataFrame({'a': ['ant', 'bee', 'cat'], 'b': ['dog', None, 'fly']})
>>> df
   a    b
0 ant  dog
1 bee None
2 cat  fly
```

```
>>> ks.isna(df)
   a    b
0 False False
1 False  True
2 False False
```

```
>>> ks.isnull(df.b)
0    False
1     True
2    False
Name: b, dtype: bool
```

## databricks.koalas.isnull

`databricks.koalas.isnull` (*obj*)

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (NaN in numeric arrays, None or NaN in object arrays).

### Parameters

**obj** [scalar or array-like] Object to check for null or missing values.

### Returns

**bool or array-like of bool** For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is missing.

See also:

**Series.isna** Detect missing values in a Series.

**Series.isnull** Detect missing values in a Series.

**DataFrame.isna** Detect missing values in a DataFrame.

**DataFrame.isnull** Detect missing values in a DataFrame.

**Index.isna** Detect missing values in an Index.

**Index.isnull** Detect missing values in an Index.

## Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> ks.isna('dog')
False
```

```
>>> ks.isna(np.nan)
True
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> ks.isna(array)
array([[False,  True, False],
       [False, False,  True]])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = ks.DataFrame({'a': ['ant', 'bee', 'cat'], 'b': ['dog', None, 'fly']})
>>> df
   a    b
0 ant  dog
1 bee None
2 cat  fly
```

```
>>> ks.isna(df)
   a    b
0 False False
1 False  True
2 False False
```

```
>>> ks.isnull(df.b)
0    False
1     True
2    False
Name: b, dtype: bool
```

## **databricks.koalas.notna**

`databricks.koalas.notna(obj)`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. NA values, such as None or numpy . NaN, get mapped to False values.

### Returns

**bool or array-like of bool** Mask of bool values for each element that indicates whether an element is not an NA value.

**See also:**

**isna** Detect missing values for an array-like object.

**Series.notna** Boolean inverse of Series.isna.

**DataFrame.notnull** Boolean inverse of DataFrame.isnull.

**Index.notna** Boolean inverse of Index.isna.

**Index.notnull** Boolean inverse of Index.isnull.

## Examples

Show which entries in a DataFrame are not NA.

```
>>> df = ks.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born      name      toy
0  5.0      NaT  Alfred      None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25      Joker
```

```
>>> df.notnull()
   age  born  name  toy
0  True False  True False
1  True  True  True  True
2 False  True  True  True
```

Show which entries in a Series are not NA.

```
>>> ser = ks.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ks.notna(ser)
0    True
1    True
2   False
dtype: bool
```

```
>>> ks.notna(ser.index)
True
```

**databricks.koalas.notnull**`databricks.koalas.notnull` (*obj*)

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. NA values, such as None or `numpy.NaN`, get mapped to False values.

**Returns**

**bool or array-like of bool** Mask of bool values for each element that indicates whether an element is not an NA value.

**See also:***isna* Detect missing values for an array-like object.*Series.notna* Boolean inverse of *Series.isna*.*DataFrame.notnull* Boolean inverse of *DataFrame.isnull*.*Index.notna* Boolean inverse of *Index.isna*.**Index.notnull** Boolean inverse of *Index.isnull*.**Examples**Show which entries in a *DataFrame* are not NA.

```
>>> df = ks.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born    name      toy
0  5.0      NaT  Alfred     None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25      Joker
```

```
>>> df.notnull()
   age  born  name  toy
0  True False  True False
1  True  True  True  True
2 False  True  True  True
```

Show which entries in a *Series* are not NA.

```
>>> ser = ks.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ks.notna(ser)
0    True
1    True
```

(continues on next page)

(continued from previous page)

```
2    False
dtype: bool
```

```
>>> ks.notna(ser.index)
True
```

### 3.2.4 Top-level dealing with datetimelike

<code>to_datetime(arg[, errors, format, unit, ...])</code>	Convert argument to datetime.
<code>date_range([start, end, periods, freq, tz, ...])</code>	Return a fixed frequency DatetimeIndex.

#### `databricks.koalas.to_datetime`

`databricks.koalas.to_datetime` (*arg*, *errors*='raise', *format*=None, *unit*=None, *infer\_datetime\_format*=False, *origin*='unix')

Convert argument to datetime.

#### Parameters

**arg** [integer, float, string, datetime, list, tuple, 1-d array, Series] or DataFrame/dict-like

**errors** [{ 'ignore', 'raise', 'coerce' }, default 'raise']

- If 'raise', then invalid parsing will raise an exception
- If 'coerce', then invalid parsing will be set as NaT
- If 'ignore', then invalid parsing will return the input

**format** [string, default None] strftime to parse time, eg “%d/%m/%Y”, note that “%f” will parse all the way up to nanoseconds.

**unit** [string, default None] unit of the arg (D,s,ms,us,ns) denote the unit, which is an integer or float number. This will be based off the origin. Example, with unit='ms' and origin='unix' (the default), this would calculate the number of milliseconds to the unix epoch start.

**infer\_datetime\_format** [boolean, default False] If True and no *format* is given, attempt to infer the format of the datetime strings, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by ~5-10x.

**origin** [scalar, default 'unix'] Define the reference date. The numeric values would be parsed as number of units (defined by *unit*) since this reference date.

- If 'unix' (or POSIX) time; origin is set to 1970-01-01.
- If 'julian', unit must be 'D', and origin is set to beginning of Julian Calendar. Julian day number 0 is assigned to the day starting at noon on January 1, 4713 BC.
- If Timestamp convertible, origin is set to Timestamp identified by origin.

#### Returns

**ret** [datetime if parsing succeeded.] Return type depends on input:

- list-like: DatetimeIndex
- Series: Series of datetime64 dtype

- scalar: Timestamp

In case when it is not possible to return designated types (e.g. when any element of input is before Timestamp.min or after Timestamp.max) return will have datetime.datetime type (or corresponding array/Series).

## Examples

Assembling a datetime from multiple columns of a DataFrame. The keys can be common abbreviations like ['year', 'month', 'day', 'minute', 'second', 'ms', 'us', 'ns']) or plurals of the same

```
>>> df = ks.DataFrame({'year': [2015, 2016],
...                     'month': [2, 3],
...                     'day': [4, 5]})
>>> ks.to_datetime(df)
0    2015-02-04
1    2016-03-05
dtype: datetime64[ns]
```

If a date does not meet the [timestamp limitations](#), passing errors='ignore' will return the original input instead of raising an exception.

Passing errors='coerce' will force an out-of-bounds date to NaT, in addition to forcing non-dates (or non-parseable dates) to NaT.

```
>>> ks.to_datetime('13000101', format='%Y%m%d', errors='ignore')
datetime.datetime(1300, 1, 1, 0, 0)
>>> ks.to_datetime('13000101', format='%Y%m%d', errors='coerce')
NaT
```

Passing infer\_datetime\_format=True can often-times speedup a parsing if its not an ISO8601 format exactly, but in a regular format.

```
>>> s = ks.Series(['3/11/2000', '3/12/2000', '3/13/2000'] * 1000)
>>> s.head()
0    3/11/2000
1    3/12/2000
2    3/13/2000
3    3/11/2000
4    3/12/2000
dtype: object
```

```
>>> import timeit
>>> timeit.timeit(
...     lambda: repr(ks.to_datetime(s, infer_datetime_format=True)),
...     number = 1)
0.35832712500000063
```

```
>>> timeit.timeit(
...     lambda: repr(ks.to_datetime(s, infer_datetime_format=False)),
...     number = 1)
0.8895321660000004
```

Using a unix epoch time

```
>>> ks.to_datetime(1490195805, unit='s')
Timestamp('2017-03-22 15:16:45')
>>> ks.to_datetime(1490195805433502912, unit='ns')
Timestamp('2017-03-22 15:16:45.433502912')
```

Using a non-unix epoch origin

```
>>> ks.to_datetime([1, 2, 3], unit='D', origin=pd.Timestamp('1960-01-01'))
DatetimeIndex(['1960-01-02', '1960-01-03', '1960-01-04'], dtype='datetime64[ns]',
               freq=None)
```

## databricks.koalas.date\_range

`databricks.koalas.date_range` (*start=None, end=None, periods=None, freq=None, tz=None, normalize=False, name=None, closed=None, \*\*kwargs*) → `databricks.koalas.indexes.datetimes.DatetimeIndex`

Return a fixed frequency `DatetimeIndex`.

### Parameters

- start** [str or datetime-like, optional] Left bound for generating dates.
- end** [str or datetime-like, optional] Right bound for generating dates.
- periods** [int, optional] Number of periods to generate.
- freq** [str or DateOffset, default 'D'] Frequency strings can have multiples, e.g. '5H'.
- tz** [str or tzinfo, optional] Time zone name for returning localized `DatetimeIndex`, for example 'Asia/Hong\_Kong'. By default, the resulting `DatetimeIndex` is timezone-naive.
- normalize** [bool, default False] Normalize start/end dates to midnight before generating date range.
- name** [str, default None] Name of the resulting `DatetimeIndex`.
- closed** [{None, 'left', 'right'}, optional] Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None, the default).
- \*\*kwargs** For compatibility. Has no effect on the result.

### Returns

**rng** [`DatetimeIndex`]

See also:

**`DatetimeIndex`** An immutable container for datetimes.

### Notes

Of the four parameters `start`, `end`, `periods`, and `freq`, exactly three must be specified. If `freq` is omitted, the resulting `DatetimeIndex` will have `periods` linearly spaced elements between `start` and `end` (closed on both sides).

To learn more about the frequency strings, please see [this link](#).

## Examples

### Specifying the values

The next four examples generate the same *DatetimeIndex*, but vary the combination of *start*, *end* and *periods*.

Specify *start* and *end*, with the default daily frequency.

```
>>> ks.date_range(start='1/1/2018', end='1/08/2018')
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
              '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08'],
              dtype='datetime64[ns]', freq=None)
```

Specify *start* and *periods*, the number of periods (days).

```
>>> ks.date_range(start='1/1/2018', periods=8)
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
              '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08'],
              dtype='datetime64[ns]', freq=None)
```

Specify *end* and *periods*, the number of periods (days).

```
>>> ks.date_range(end='1/1/2018', periods=8)
DatetimeIndex(['2017-12-25', '2017-12-26', '2017-12-27', '2017-12-28',
              '2017-12-29', '2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq=None)
```

Specify *start*, *end*, and *periods*; the frequency is generated automatically (linearly spaced).

```
>>> ks.date_range(
...     start='2018-04-24', end='2018-04-27', periods=3
... )
DatetimeIndex(['2018-04-24 00:00:00', '2018-04-25 12:00:00',
              '2018-04-27 00:00:00'],
              dtype='datetime64[ns]', freq=None)
```

### Other Parameters

Changed the *freq* (frequency) to 'M' (month end frequency).

```
>>> ks.date_range(start='1/1/2018', periods=5, freq='M')
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31', '2018-04-30',
              '2018-05-31'],
              dtype='datetime64[ns]', freq=None)
```

Multiples are allowed

```
>>> ks.date_range(start='1/1/2018', periods=5, freq='3M')
DatetimeIndex(['2018-01-31', '2018-04-30', '2018-07-31', '2018-10-31',
              '2019-01-31'],
              dtype='datetime64[ns]', freq=None)
```

*freq* can also be specified as an Offset object.

```
>>> ks.date_range(
...     start='1/1/2018', periods=5, freq=pd.offsets.MonthEnd(3)
... )
DatetimeIndex(['2018-01-31', '2018-04-30', '2018-07-31', '2018-10-31',
```

(continues on next page)



(continued from previous page)

```
'2019-01-31'],
dtype='datetime64[ns]', freq=None)
```

*closed* controls whether to include *start* and *end* that are on the boundary. The default includes boundary points on either end.

```
>>> ks.date_range(
...     start='2017-01-01', end='2017-01-04', closed=None
... )
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03', '2017-01-04'],
              dtype='datetime64[ns]', freq=None)
```

Use *closed='left'* to exclude *end* if it falls on the boundary.

```
>>> ks.date_range(
...     start='2017-01-01', end='2017-01-04', closed='left'
... )
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03'], dtype='datetime64[ns]',
              ↪freq=None)
```

Use *closed='right'* to exclude *start* if it falls on the boundary.

```
>>> ks.date_range(
...     start='2017-01-01', end='2017-01-04', closed='right'
... )
DatetimeIndex(['2017-01-02', '2017-01-03', '2017-01-04'], dtype='datetime64[ns]',
              ↪freq=None)
```

## 3.3 Series

### 3.3.1 Constructor

---

```
Series([data, index, dtype, name, copy, ...])
```

---

Koalas Series that corresponds to pandas Series logically.

---

#### `databricks.koalas.Series`

```
class databricks.koalas.Series (data=None, index=None, dtype=None, name=None,
                                copy=False, fastpath=False)
```

Koalas Series that corresponds to pandas Series logically. This holds Spark Column internally.

#### Variables

- `_internal` – an internal immutable Frame to manage metadata.
- `_kdf` – Parent's Koalas DataFrame

#### Parameters

**data** [array-like, dict, or scalar value, pandas Series] Contains data stored in Series If data is a dict, argument order is maintained for Python 3.6 and later. Note that if *data* is a pandas Series, other arguments should not be used.

**index** [array-like or Index (1d)] Values must be hashable and have the same length as *data*. Non-unique index values are allowed. Will default to RangeIndex (0, 1, 2, ..., n) if not provided. If both a dict and index sequence are used, the index will override the keys found in the dict.

**dtype** [numpy.dtype or None] If None, dtype will be inferred

**copy** [boolean, default False] Copy input data

**\_\_init\_\_** (*data=None, index=None, dtype=None, name=None, copy=False, fastpath=False*)  
Initialize self. See help(type(self)) for accurate signature.

## Methods

<code>__init__</code> ([ <i>data, index, dtype, name, copy, ...</i> ])	Initialize self.
<code>abs</code> ()	Return a Series/DataFrame with absolute numeric value of each element.
<code>add</code> (other)	Return Addition of series and other, element-wise (binary operator +).
<code>add_prefix</code> (prefix)	Prefix labels with string <i>prefix</i> .
<code>add_suffix</code> (suffix)	Suffix labels with string <i>suffix</i> .
<code>agg</code> (func)	Aggregate using one or more operations over the specified axis.
<code>aggregate</code> (func)	Aggregate using one or more operations over the specified axis.
<code>alias</code> (name)	An alias for <code>Series.rename()</code> .
<code>align</code> (other[, join, axis, copy])	Align two objects on their axes with the specified join method.
<code>all</code> ([axis])	Return whether all elements are True.
<code>any</code> ([axis])	Return whether any element is True.
<code>append</code> (to_append[, ignore_index, ...])	Concatenate two or more Series.
<code>apply</code> (func[, args])	Invoke function on values of Series.
<code>argmax</code> ()	Return int position of the largest value in the Series.
<code>argmin</code> ()	Return int position of the smallest value in the Series.
<code>argsort</code> ()	Return the integer indices that would sort the Series values.
<code>asof</code> (where)	Return the last row(s) without any NaNs before <i>where</i> .
<code>astype</code> (dtype)	Cast a Koalas object to a specified dtype <i>dtype</i> .
<code>at_time</code> (time[, asof, axis])	Select values at particular time of day (e.g., 9:30AM).
<code>backfill</code> ([axis, inplace, limit])	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='bfill'</code> .
<code>between</code> (left, right[, inclusive])	Return boolean Series equivalent to <code>left &lt;= series &lt;= right</code> .
<code>between_time</code> (start_time, end_time[, ...])	Select values between particular times of the day (e.g., 9:00-9:30 AM).
<code>bfill</code> ([axis, inplace, limit])	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='bfill'</code> .
<code>bool</code> ()	Return the bool of a single element in the current object.
<code>clip</code> ([lower, upper])	Trim values at input threshold(s).

continues on next page

Table 18 – continued from previous page

<code>combine_first(other)</code>	Combine Series values, choosing the calling Series's values first.
<code>compare(other[, keep_shape, keep_equal])</code>	Compare to another Series and show the differences.
<code>copy([deep])</code>	Make a copy of this object's indices and data.
<code>corr(other[, method])</code>	Compute correlation with <i>other</i> Series, excluding missing values.
<code>count([axis, numeric_only])</code>	Count non-NA cells for each column.
<code>cummax([skipna])</code>	Return cumulative maximum over a DataFrame or Series axis.
<code>cummin([skipna])</code>	Return cumulative minimum over a DataFrame or Series axis.
<code>cumprod([skipna])</code>	Return cumulative product over a DataFrame or Series axis.
<code>cumsum([skipna])</code>	Return cumulative sum over a DataFrame or Series axis.
<code>describe([percentiles])</code>	Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
<code>diff([periods])</code>	First discrete difference of element.
<code>div(other)</code>	Return Floating division of series and other, element-wise (binary operator <code>/</code> ).
<code>divide(other)</code>	Return Floating division of series and other, element-wise (binary operator <code>/</code> ).
<code>divmod(other)</code>	Return Integer division and modulo of series and other, element-wise (binary operator <code>divmod</code> ).
<code>dot(other)</code>	Compute the dot product between the Series and the columns of other.
<code>drop([labels, index, level])</code>	Return Series with specified index labels removed.
<code>drop_duplicates([keep, inplace])</code>	Return Series with duplicate values removed.
<code>droplevel(level)</code>	Return Series with requested index level(s) removed.
<code>dropna([axis, inplace])</code>	Return a new Series with missing values removed.
<code>eq(other)</code>	Compare if the current value is equal to the other.
<code>equals(other)</code>	Compare if the current value is equal to the other.
<code>expanding([min_periods])</code>	Provide expanding transformations.
<code>explode()</code>	Transform each element of a list-like to a row.
<code>factorize([sort, na_sentinel])</code>	Encode the object as an enumerated type or categorical variable.
<code>ffill([axis, inplace, limit])</code>	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='ffill'</code> .
<code>fillna([value, method, axis, inplace, limit])</code>	Fill NA/NaN values.
<code>filter([items, like, regex, axis])</code>	Subset rows or columns of dataframe according to labels in the specified index.
<code>first(offset)</code>	Select first periods of time series data based on a date offset.
<code>first_valid_index()</code>	Retrieves the index of the first valid value.
<code>floordiv(other)</code>	Return Integer division of series and other, element-wise (binary operator <code>//</code> ).
<code>ge(other)</code>	Compare if the current value is greater than or equal to the other.
<code>get(key[, default])</code>	Get item from object for given key (DataFrame column, Panel slice, etc.).

continues on next page

Table 18 – continued from previous page

<code>get_dtype_counts()</code>	Return counts of unique dtypes in this object.
<code>groupby(by[, axis, as_index, dropna])</code>	Group DataFrame or Series using a Series of columns.
<code>gt(other)</code>	Compare if the current value is greater than the other.
<code>head([n])</code>	Return the first <i>n</i> rows.
<code>hist([bins])</code>	Draw one histogram of the DataFrame's columns.
<code>idxmax([skipna])</code>	Return the row label of the maximum value.
<code>idxmin([skipna])</code>	Return the row label of the minimum value.
<code>isin(values)</code>	Check whether <i>values</i> are contained in Series or Index.
<code>isna()</code>	Detect existing (non-missing) values.
<code>isnull()</code>	Detect existing (non-missing) values.
<code>item()</code>	Return the first element of the underlying data as a Python scalar.
<code>items()</code>	This is an alias of <code>iteritems</code> .
<code>iteritems()</code>	Lazily iterate over (index, value) tuples.
<code>keys()</code>	Return alias for index.
<code>kurt([axis, numeric_only])</code>	Return unbiased kurtosis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>kurtosis([axis, numeric_only])</code>	Return unbiased kurtosis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>last(offset)</code>	Select final periods of time series data based on a date offset.
<code>last_valid_index()</code>	Return index for last non-NA/null value.
<code>le(other)</code>	Compare if the current value is less than or equal to the other.
<code>lt(other)</code>	Compare if the current value is less than the other.
<code>mad()</code>	Return the mean absolute deviation of values.
<code>map(arg)</code>	Map values of Series according to input correspondence.
<code>mask(cond[, other])</code>	Replace values where the condition is True.
<code>max([axis, numeric_only])</code>	Return the maximum of the values.
<code>mean([axis, numeric_only])</code>	Return the mean of the values.
<code>median([axis, numeric_only, accuracy])</code>	Return the median of the values for the requested axis.
<code>min([axis, numeric_only])</code>	Return the minimum of the values.
<code>mod(other)</code>	Return Modulo of series and other, element-wise (binary operator %).
<code>mode([dropna])</code>	Return the mode(s) of the dataset.
<code>mul(other)</code>	Return Multiplication of series and other, element-wise (binary operator *).
<code>multiply(other)</code>	Return Multiplication of series and other, element-wise (binary operator *).
<code>ne(other)</code>	Compare if the current value is not equal to the other.
<code>nlargest([n])</code>	Return the largest <i>n</i> elements.
<code>notna()</code>	Detect existing (non-missing) values.
<code>notnull()</code>	Detect existing (non-missing) values.
<code>nsmallest([n])</code>	Return the smallest <i>n</i> elements.
<code>nunique([dropna, approx, rsd])</code>	Return number of unique elements in the object.
<code>pad([axis, inplace, limit])</code>	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='ffill'</code> .

continues on next page

Table 18 – continued from previous page

<code>pct_change([periods])</code>	Percentage change between the current and a prior element.
<code>pipe(func, *args, **kwargs)</code>	Apply <code>func(self, *args, **kwargs)</code> .
<code>pop(item)</code>	Return item and drop from series.
<code>pow(other)</code>	Return Exponential power of series of series and other, element-wise (binary operator <code>**</code> ).
<code>prod([axis, numeric_only, min_count])</code>	Return the product of the values.
<code>product([axis, numeric_only, min_count])</code>	Return the product of the values.
<code>quantile([q, accuracy])</code>	Return value at the given quantile.
<code>radd(other)</code>	Return Reverse Addition of series and other, element-wise (binary operator <code>+</code> ).
<code>rank([method, ascending])</code>	Compute numerical data ranks (1 through n) along axis.
<code>rdiv(other)</code>	Return Reverse Floating division of series and other, element-wise (binary operator <code>/</code> ).
<code>rdivmod(other)</code>	Return Integer division and modulo of series and other, element-wise (binary operator <code>rdivmod</code> ).
<code>reindex([index, fill_value])</code>	Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
<code>reindex_like(other)</code>	Return a Series with matching indices as other object.
<code>rename([index])</code>	Alter Series name.
<code>rename_axis([mapper, index, inplace])</code>	Set the name of the axis for the index or columns.
<code>repeat(repeats)</code>	Repeat elements of a Series.
<code>replace([to_replace, value, regex])</code>	Replace values given in <code>to_replace</code> with value.
<code>reset_index([level, drop, name, inplace])</code>	Generate a new DataFrame or Series with the index reset.
<code>rfloordiv(other)</code>	Return Reverse Integer division of series and other, element-wise (binary operator <code>//</code> ).
<code>rmod(other)</code>	Return Reverse Modulo of series and other, element-wise (binary operator <code>%</code> ).
<code>rmul(other)</code>	Return Reverse Multiplication of series and other, element-wise (binary operator <code>*</code> ).
<code>rolling(window[, min_periods])</code>	Provide rolling transformations.
<code>round([decimals])</code>	Round each value in a Series to the given number of decimals.
<code>rpow(other)</code>	Return Reverse Exponential power of series and other, element-wise (binary operator <code>**</code> ).
<code>rsub(other)</code>	Return Reverse Subtraction of series and other, element-wise (binary operator <code>-</code> ).
<code>rtruediv(other)</code>	Return Reverse Floating division of series and other, element-wise (binary operator <code>/</code> ).
<code>sample([n, frac, replace, random_state])</code>	Return a random sample of items from an axis of object.
<code>sem([axis, ddof, numeric_only])</code>	Return unbiased standard error of the mean over requested axis.
<code>shift([periods, fill_value])</code>	Shift Series/Index by desired number of periods.
<code>skew([axis, numeric_only])</code>	Return unbiased skew normalized by N-1.
<code>sort_index([axis, level, ascending, ...])</code>	Sort object by labels (along an axis)
<code>sort_values([ascending, inplace, na_position])</code>	Sort by the values.

continues on next page

Table 18 – continued from previous page

<code>squeeze([axis])</code>	Squeeze 1 dimensional axis objects into scalars.
<code>std([axis, ddof, numeric_only])</code>	Return sample standard deviation.
<code>sub(other)</code>	Return Subtraction of series and other, element-wise (binary operator -).
<code>subtract(other)</code>	Return Subtraction of series and other, element-wise (binary operator -).
<code>sum([axis, numeric_only, min_count])</code>	Return the sum of the values.
<code>swapaxes(i, j[, copy])</code>	Interchange axes and swap values axes appropriately.
<code>swaplevel([i, j, copy])</code>	Swap levels i and j in a MultiIndex.
<code>tail([n])</code>	Return the last <i>n</i> rows.
<code>take(indices)</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>toPandas()</code>	Return a pandas Series.
<code>to_clipboard([excel, sep])</code>	Copy object to the system clipboard.
<code>to_csv([path, sep, na_rep, columns, header, ...])</code>	Write object to a comma-separated values (csv) file.
<code>to_dataframe([name])</code>	Convert Series to DataFrame.
<code>to_dict([into])</code>	Convert Series to {label -> value} dict or dict-like object.
<code>to_excel(excel_writer[, sheet_name, na_rep, ...])</code>	Write object to an Excel sheet.
<code>to_frame([name])</code>	Convert Series to DataFrame.
<code>to_json([path, compression, num_files, ...])</code>	Convert the object to a JSON string.
<code>to_latex([buf, columns, col_space, header, ...])</code>	Render an object to a LaTeX tabular environment table.
<code>to_list()</code>	Return a list of the values.
<code>to_markdown([buf, mode])</code>	Print Series or DataFrame in Markdown-friendly format.
<code>to_numpy()</code>	A NumPy ndarray representing the values in this DataFrame or Series.
<code>to_pandas()</code>	Return a pandas Series.
<code>to_string([buf, na_rep, float_format, ...])</code>	Render a string representation of the Series.
<code>tolist()</code>	Return a list of the values.
<code>transform(func[, axis])</code>	Call <code>func</code> producing the same type as <i>self</i> with transformed values and that has the same axis length as input.
<code>transform_batch(func, *args, **kwargs)</code>	Transform the data with the function that takes pandas Series and outputs pandas Series.
<code>transpose(*args, **kwargs)</code>	Return the transpose, which is by definition self.
<code>truediv(other)</code>	Return Floating division of series and other, element-wise (binary operator /).
<code>truncate([before, after, axis, copy])</code>	Truncate a Series or DataFrame before and after some index value.
<code>unique()</code>	Return unique values of Series object.
<code>unstack([level])</code>	Unstack, a.k.a.
<code>update(other)</code>	Modify Series in place using non-NA values from passed Series.
<code>value_counts([normalize, sort, ascending, ...])</code>	Return a Series containing counts of unique values.
<code>var([axis, ddof, numeric_only])</code>	Return unbiased variance.
<code>where(cond[, other])</code>	Replace values where the condition is False.
<code>xs(key[, level])</code>	Return cross-section from the Series.

**Attributes**

<i>T</i>	Return the transpose, which is by definition self.
<i>at</i>	Access a single value for a row/column label pair.
<i>axes</i>	Return a list of the row axis labels.
<i>dtype</i>	Return the dtype object of the underlying data.
<i>dtypes</i>	Return the dtype object of the underlying data.
<i>empty</i>	Returns true if the current object is empty.
<i>hasnans</i>	Return True if it has any missing values.
<i>iat</i>	Access a single value for a row/column pair by integer position.
<i>iloc</i>	Purely integer-location based indexing for selection by position.
<i>index</i>	The index (axis labels) Column of the Series.
<i>is_monotonic</i>	Return boolean if values in the object are monotonically increasing.
<i>is_monotonic_decreasing</i>	Return boolean if values in the object are monotonically decreasing.
<i>is_monotonic_increasing</i>	Return boolean if values in the object are monotonically increasing.
<i>is_series</i>	
<i>is_unique</i>	Return boolean if values in the object are unique
<i>loc</i>	Access a group of rows and columns by label(s) or a boolean Series.
<i>name</i>	Return name of the Series.
<i>ndim</i>	Return an int representing the number of array dimensions.
<i>shape</i>	Return a tuple of the shape of the underlying data.
<i>size</i>	Return an int representing the number of elements in this object.
<i>spark_column</i>	Spark Column object representing the Series/Index.
<i>spark_type</i>	Returns the data type as defined by Spark, as a Spark DataType object.
<i>values</i>	Return a Numpy representation of the DataFrame or the Series.

**3.3.2 Attributes**

<i>Series.index</i>	The index (axis labels) Column of the Series.
<i>Series.dtype</i>	Return the dtype object of the underlying data.
<i>Series.dtypes</i>	Return the dtype object of the underlying data.
<i>Series.ndim</i>	Return an int representing the number of array dimensions.
<i>Series.name</i>	Return name of the Series.
<i>Series.shape</i>	Return a tuple of the shape of the underlying data.
<i>Series.axes</i>	Return a list of the row axis labels.
<i>Series.size</i>	Return an int representing the number of elements in this object.
<i>Series.empty</i>	Returns true if the current object is empty.

continues on next page

Table 20 – continued from previous page

<code>Series.T</code>	Return the transpose, which is by definition self.
<code>Series.hasnans</code>	Return True if it has any missing values.
<code>Series.values</code>	Return a Numpy representation of the DataFrame or the Series.

**databricks.koalas.Series.index****property** `Series.index`

The index (axis labels) Column of the Series.

See also:

[\*Index\*](#)

**databricks.koalas.Series.dtype****property** `Series.dtype`

Return the dtype object of the underlying data.

**Examples**

```
>>> s = ks.Series([1, 2, 3])
>>> s.dtype
dtype('int64')
```

```
>>> s = ks.Series(list('abc'))
>>> s.dtype
dtype('O')
```

```
>>> s = ks.Series(pd.date_range('20130101', periods=3))
>>> s.dtype
dtype('<M8[ns]')
```

```
>>> s.rename("a").to_frame().set_index("a").index.dtype
dtype('<M8[ns]')
```

**databricks.koalas.Series.dtypes****property** `Series.dtypes`

Return the dtype object of the underlying data.

```
>>> s = ks.Series(list('abc'))
>>> s.dtype == s.dtypes
True
```



**databricks.koalas.Series.ndim****property** `Series.ndim`

Return an int representing the number of array dimensions.

Return 1 for Series / Index / MultiIndex.

**Examples**

For Series

```
>>> s = ks.Series([None, 1, 2, 3, 4], index=[4, 5, 2, 1, 8])
>>> s.ndim
1
```

For Index

```
>>> s.index.ndim
1
```

For MultiIndex

```
>>> midx = pd.MultiIndex([['lama', 'cow', 'falcon'],
...                       ['speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [1, 1, 1, 1, 1, 2, 1, 2, 2]])
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3], index=midx)
>>> s.index.ndim
1
```

**databricks.koalas.Series.name****property** `Series.name`

Return name of the Series.

**databricks.koalas.Series.shape****property** `Series.shape`

Return a tuple of the shape of the underlying data.

**databricks.koalas.Series.axes****property** `Series.axes`

Return a list of the row axis labels.

## Examples

```
>>> kser = ks.Series([1, 2, 3])
>>> kser.axes
[Int64Index([0, 1, 2], dtype='int64')]
```

## databricks.koalas.Series.size

### property Series.size

Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

## Examples

```
>>> s = ks.Series({'a': 1, 'b': 2, 'c': None})
>>> s.size
3
```

```
>>> df = ks.DataFrame({'col1': [1, 2, None], 'col2': [3, 4, None]})
>>> df.size
6
```

```
>>> df = ks.DataFrame(index=[1, 2, None])
>>> df.size
0
```

## databricks.koalas.Series.empty

### property Series.empty

Returns true if the current object is empty. Otherwise, returns false.

```
>>> ks.range(10).id.empty
False
```

```
>>> ks.range(0).id.empty
True
```

```
>>> ks.DataFrame({}, index=list('abc')).index.empty
False
```

**databricks.koalas.Series.T****property** Series.**T**

Return the transpose, which is by definition self.

**Examples**

It returns the same object as the transpose of the given series object, which is by definition self.

```
>>> s = ks.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.transpose()
0    1
1    2
2    3
dtype: int64
```

**databricks.koalas.Series.hasnans****property** Series.**hasnans**

Return True if it has any missing values. Otherwise, it returns False.

```
>>> ks.DataFrame({}, index=list('abc')).index.hasnans
False
```

```
>>> ks.Series(['a', None]).hasnans
True
```

```
>>> ks.Series([1.0, 2.0, np.nan]).hasnans
True
```

```
>>> ks.Series([1, 2, 3]).hasnans
False
```

```
>>> (ks.Series([1.0, 2.0, np.nan]) + 1).hasnans
True
```

```
>>> ks.Series([1, 2, 3]).rename("a").to_frame().set_index("a").index.hasnans
False
```

## `databricks.koalas.Series.values`

### property `Series.values`

Return a Numpy representation of the DataFrame or the Series.

**Warning:** We recommend using `DataFrame.to_numpy()` or `Series.to_numpy()` instead.

---

**Note:** This method should only be used if the resulting NumPy ndarray is expected to be small, as all the data is loaded into the driver's memory.

---

### Returns

`numpy.ndarray`

### Examples

A DataFrame where all columns are the same type (e.g., `int64`) results in an array of the same type.

```
>>> df = ks.DataFrame({'age':    [ 3,   29],
...                   'height': [94, 170],
...                   'weight': [31, 115]})
>>> df
   age  height  weight
0    3     94     31
1   29    170    115
>>> df.dtypes
age      int64
height  int64
weight  int64
dtype: object
>>> df.values
array([[ 3,  94,  31],
       [29, 170, 115]])
```

A DataFrame with mixed type columns (e.g., `str/object`, `int64`, `float32`) results in an ndarray of the broadest type that accommodates these mixed types (e.g., `object`).

```
>>> df2 = ks.DataFrame([('parrot',    24.0, 'second'),
...                    ('lion',     80.5, 'first'),
...                    ('monkey', np.nan, None)],
...                    columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name      object
max_speed  float64
rank      object
dtype: object
>>> df2.values
array([('parrot', 24.0, 'second'),
      ('lion', 80.5, 'first'),
      ('monkey', nan, None)], dtype=object)
```

For Series,

```
>>> ks.Series([1, 2, 3]).values
array([1, 2, 3])
```

```
>>> ks.Series(list('aabc')).values
array(['a', 'a', 'b', 'c'], dtype=object)
```

### 3.3.3 Conversion

<code>Series.astype(dtype)</code>	Cast a Koalas object to a specified dtype <code>dtype</code> .
<code>Series.copy([deep])</code>	Make a copy of this object's indices and data.
<code>Series.bool()</code>	Return the bool of a single element in the current object.

#### `databricks.koalas.Series.astype`

`Series.astype(dtype: Union[str, type, numpy.dtype, pandas.core.dtypes.base.ExtensionDtype]) → Union[Index, Series]`

Cast a Koalas object to a specified dtype `dtype`.

##### Parameters

**dtype** [data type] Use a `numpy.dtype` or Python type to cast entire pandas object to the same type.

##### Returns

**casted** [same type as caller]

See also:

[`to\_datetime`](#) Convert argument to datetime.

#### Examples

```
>>> ser = ks.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
```

```
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

```
>>> ser.rename("a").to_frame().set_index("a").index.astype('int64')
Int64Index([1, 2], dtype='int64', name='a')
```

### `databricks.koalas.Series.copy`

`Series.copy(deep=None) → databricks.koalas.series.Series`

Make a copy of this object's indices and data.

#### Parameters

**deep** [None] this parameter is not supported but just dummy parameter to match pandas.

#### Returns

**copy** [Series]

### Examples

```
>>> s = ks.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

### `databricks.koalas.Series.bool`

`Series.bool() → bool`

Return the bool of a single element in the current object.

This must be a boolean scalar value, either True or False. Raise a `ValueError` if the object does not have exactly 1 element, or that element is not boolean

#### Returns

**bool**

### Examples

```
>>> ks.DataFrame({'a': [True]}).bool()
True
```

```
>>> ks.Series([False]).bool()
False
```

If there are non-boolean or multiple values exist, it raises an exception in all cases as below.

```
>>> ks.DataFrame({'a': ['a']}).bool()
Traceback (most recent call last):
...
ValueError: bool cannot act on a non-boolean single element DataFrame
```

```
>>> ks.DataFrame({'a': [True], 'b': [False]}).bool()
Traceback (most recent call last):
...
ValueError: The truth value of a DataFrame is ambiguous. Use a.empty, a.bool(),
a.item(), a.any() or a.all().
```

```
>>> ks.Series([1]).bool()
Traceback (most recent call last):
...
ValueError: bool cannot act on a non-boolean single element DataFrame
```

### 3.3.4 Indexing, iteration

<code>Series.at</code>	Access a single value for a row/column label pair.
<code>Series.iat</code>	Access a single value for a row/column pair by integer position.
<code>Series.loc</code>	Access a group of rows and columns by label(s) or a boolean Series.
<code>Series.iloc</code>	Purely integer-location based indexing for selection by position.
<code>Series.keys()</code>	Return alias for index.
<code>Series.pop(item)</code>	Return item and drop from series.
<code>Series.items()</code>	This is an alias of <code>iteritems</code> .
<code>Series.iteritems()</code>	Lazily iterate over (index, value) tuples.
<code>Series.item()</code>	Return the first element of the underlying data as a Python scalar.
<code>Series.xs(key[, level])</code>	Return cross-section from the Series.
<code>Series.get(key[, default])</code>	Get item from object for given key (DataFrame column, Panel slice, etc.).

#### `databricks.koalas.Series.at`

##### **property** `Series.at`

Access a single value for a row/column label pair. If the index is not unique, all matching pairs are returned as an array. Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get a single value in a DataFrame or Series.

---

**Note:** Unlike `pandas`, Koalas only allows using `at` to get values but not to set them.

---



---

**Note:** Warning: If `row_index` matches a lot of rows, large amounts of data will be fetched, potentially causing your machine to run out of memory.

---

##### **Raises**

**KeyError** When label does not exist in DataFrame

## Examples

```
>>> kdf = ks.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                     index=[4, 5, 5], columns=['A', 'B', 'C'])
>>> kdf
   A  B  C
4  0  2  3
5  0  4  1
5 10 20 30
```

Get value at specified row/column pair

```
>>> kdf.at[4, 'B']
2
```

Get array if an index occurs multiple times

```
>>> kdf.at[5, 'B']
array([ 4, 20])
```

## `databricks.koalas.Series.iat`

### property `Series.iat`

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

#### Raises

**KeyError** When label does not exist in DataFrame

## Examples

```
>>> df = ks.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                    columns=['A', 'B', 'C'])
>>> df
   A  B  C
0  0  2  3
1  0  4  1
2 10 20 30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Get value within a series

```
>>> kser = ks.Series([1, 2, 3], index=[10, 20, 30])
>>> kser
10    1
20    2
30    3
dtype: int64
```



```
>>> kser.iat[1]
2
```

## databricks.koalas.Series.loc

### property Series.loc

Access a group of rows and columns by label(s) or a boolean Series.

.loc[] is primarily label based, but may also be used with a conditional boolean Series derived from the DataFrame or Series.

Allowed inputs are:

- A single label, e.g. 5 or 'a', (note that 5 is interpreted as a *label* of the index, and **never** as an integer position along the index) for column selection.
- A list or array of labels, e.g. ['a', 'b', 'c'].
- A slice object with labels, e.g. 'a':'f'.
- A conditional boolean Series derived from the DataFrame or Series
- A boolean array of the same length as the column axis being sliced, e.g. [True, False, True].
- An alignable boolean pandas Series to the column axis being sliced. The index of the key will be aligned before masking.

Not allowed inputs which pandas allows are:

- A boolean array of the same length as the row axis being sliced, e.g. [True, False, True].
- A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

---

**Note:** MultiIndex is not supported yet.

---



---

**Note:** Note that contrary to usual python slices, **both** the start and the stop are included, and the step of the slice is not allowed.

---



---

**Note:** With a list or array of labels for row selection, Koalas behaves as a filter without reordering by the labels.

---

**See also:**

[\*Series.loc\*](#) Access group of values using labels.

## Examples

### Getting values

```
>>> df = ks.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=['cobra', 'viper', 'sidewinder'],
...                    columns=['max_speed', 'shield'])
>>> df
```

	max_speed	shield
cobra	1	2
viper	4	5
sidewinder	7	8

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[ ]]` returns a DataFrame. Also note that Koalas behaves just a filter without reordering by the labels.

```
>>> df.loc[['viper', 'sidewinder']]
      max_speed  shield
viper         4       5
sidewinder    7       8
```

```
>>> df.loc[['sidewinder', 'viper']]
      max_speed  shield
viper         4       5
sidewinder    7       8
```

Single label for column.

```
>>> df.loc['cobra', 'shield']
2
```

List of labels for row.

```
>>> df.loc[['cobra'], 'shield']
cobra    2
Name: shield, dtype: int64
```

List of labels for column.

```
>>> df.loc['cobra', ['shield']]
shield    2
Name: cobra, dtype: int64
```

List of labels for both row and column.

```
>>> df.loc[['cobra'], ['shield']]
      shield
cobra     2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra      1
viper      4
Name: max_speed, dtype: int64
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
      max_speed  shield
sidewinder      7      8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
      max_speed
sidewinder      7
```

A boolean array of the same length as the column axis being sliced.

```
>>> df.loc[:, [False, True]]
      shield
cobra      2
viper      5
sidewinder  8
```

An alignable boolean Series to the column axis being sliced.

```
>>> df.loc[:, pd.Series([False, True], index=['max_speed', 'shield'])]
      shield
cobra      2
viper      5
sidewinder  8
```

### Setting values

Setting value for all items matching the list of labels.

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
      max_speed  shield
cobra          1      2
viper          4     50
sidewinder      7     50
```

Setting value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
      max_speed  shield
cobra         10     10
viper          4     50
sidewinder      7     50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
```

(continues on next page)

(continued from previous page)

	max_speed	shield
cobra	30	10
viper	30	50
sidewinder	30	50

Set value for an entire list of columns

```
>>> df.loc[:, ['max_speed', 'shield']] = 100
>>> df
```

	max_speed	shield
cobra	100	100
viper	100	100
sidewinder	100	100

Set value with Series

```
>>> df.loc[:, 'shield'] = df['shield'] * 2
>>> df
```

	max_speed	shield
cobra	100	200
viper	100	200
sidewinder	100	200

### Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = ks.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=[7, 8, 9],
...                    columns=['max_speed', 'shield'])
>>> df
```

	max_speed	shield
7	1	2
8	4	5
9	7	8

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
```

	max_speed	shield
7	1	2
8	4	5
9	7	8

## databricks.koalas.Series.iloc

### property Series.iloc

Purely integer-location based indexing for selection by position.

.iloc[] is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a conditional boolean Series.

Allowed inputs are:

- An integer for column selection, e.g. 5.
- A list or array of integers for row selection with distinct index values, e.g. [3, 4, 0]

- A list or array of integers for column selection, e.g. [4, 3, 0].
- A boolean array for column selection.
- A slice object with ints for row and column selection, e.g. 1:7.

Not allowed inputs which pandas allows are:

- A list or array of integers for row selection with duplicated indexes, e.g. [4, 4, 0].
- A boolean array for row selection.
- A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See also:

**`DataFrame.loc`** Purely label-location based indexer for selection by label.

**`Series.iloc`** Purely integer-location based indexing for selection by position.

## Examples

```
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000}]
>>> df = ks.DataFrame(mydict, columns=['a', 'b', 'c', 'd'])
>>> df
   a    b    c    d
0   1    2    3    4
1  100  200  300  400
2 1000 2000 3000 4000
```

### Indexing just the rows

A scalar integer for row selection.

```
>>> df.iloc[1]
a    100
b    200
c    300
d    400
Name: 1, dtype: int64
```

```
>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
```

With a *slice* object.

```
>>> df.iloc[:3]
   a    b    c    d
0   1    2    3    4
1  100  200  300  400
2 1000 2000 3000 4000
```

### Indexing both axes

You can mix the indexer types for the index and columns. Use `:` to select the entire axis.

With scalar integers.

```
>>> df.iloc[:, 1]
0      2
Name: b, dtype: int64
```

With lists of integers.

```
>>> df.iloc[:, 2, [1, 3]]
      b      d
0      2      4
1    200    400
```

With *slice* objects.

```
>>> df.iloc[:, 0:3]
      a      b      c
0      1      2      3
1    100    200    300
```

With a boolean array whose length matches the columns.

```
>>> df.iloc[:, [True, False, True, False]]
      a      c
0      1      3
1    100    300
2   1000   3000
```

### Setting values

Setting value for all items matching the list of labels.

```
>>> df.iloc[[1, 2], [1]] = 50
>>> df
      a      b      c      d
0      1      2      3      4
1    100    50    300    400
2   1000    50   3000   4000
```

Setting value for an entire row

```
>>> df.iloc[0] = 10
>>> df
      a      b      c      d
0     10    10     10     10
1    100    50    300    400
2   1000    50   3000   4000
```

Set value for an entire column

```
>>> df.iloc[:, 2] = 30
>>> df
      a      b      c      d
0     10    10    30     10
```

(continues on next page)

(continued from previous page)

```
1  100  50  30  400
2 1000  50  30 4000
```

Set value for an entire list of columns

```
>>> df.iloc[:, [2, 3]] = 100
>>> df
   a  b  c  d
0  10 10 100 100
1  100 50 100 100
2 1000 50 100 100
```

Set value with Series

```
>>> df.iloc[:, 3] = df.iloc[:, 3] * 2
>>> df
   a  b  c  d
0  10 10 100 200
1  100 50 100 200
2 1000 50 100 200
```

### `databricks.koalas.Series.keys`

`Series.keys()` → `databricks.koalas.indexes.base.Index`

Return alias for index.

#### Returns

**Index** Index of the Series.

### Examples

```
>>> midx = pd.MultiIndex([['lama', 'cow', 'falcon'],
...                       ['speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> kser = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3], index=midx)
```

```
>>> kser.keys()
MultiIndex([( 'lama', 'speed'),
            ( 'lama', 'weight'),
            ( 'lama', 'length'),
            ( 'cow', 'speed'),
            ( 'cow', 'weight'),
            ( 'cow', 'length'),
            ('falcon', 'speed'),
            ('falcon', 'weight'),
            ('falcon', 'length')],
           )
```

**databricks.koalas.Series.pop**

`Series.pop(item)` → Union[databricks.koalas.series.Series, int, float, str, bytes, decimal.Decimal, date-time.date, None]  
Return item and drop from series.

**Parameters**

**item** [str] Label of index to be popped.

**Returns**

**Value that is popped from series.**

**Examples**

```
>>> s = ks.Series(data=np.arange(3), index=['A', 'B', 'C'])
>>> s
A    0
B    1
C    2
dtype: int64
```

```
>>> s.pop('A')
0
```

```
>>> s
B    1
C    2
dtype: int64
```

```
>>> s = ks.Series(data=np.arange(3), index=['A', 'A', 'C'])
>>> s
A    0
A    1
C    2
dtype: int64
```

```
>>> s.pop('A')
A    0
A    1
dtype: int64
```

```
>>> s
C    2
dtype: int64
```

**Also support for MultiIndex**

```
>>> midx = pd.MultiIndex([[ 'lama', 'cow', 'falcon'],
...                       [ 'speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3],
...               index=midx)
>>> s
```

(continues on next page)



(continued from previous page)

```

lama    speed    45.0
        weight   200.0
        length   1.2
cow     speed    30.0
        weight   250.0
        length   1.5
falcon  speed    320.0
        weight   1.0
        length   0.3
dtype: float64

```

```

>>> s.pop('lama')
speed    45.0
weight   200.0
length   1.2
dtype: float64

```

```

>>> s
cow     speed    30.0
        weight   250.0
        length   1.5
falcon  speed    320.0
        weight   1.0
        length   0.3
dtype: float64

```

Also support for MultiIndex with several indexes.

```

>>> midx = pd.MultiIndex([[ 'a', 'b', 'c'],
...                       [ 'lama', 'cow', 'falcon'],
...                       [ 'speed', 'weight', 'length']],
...                       [[0, 0, 0, 0, 0, 0, 1, 1, 1],
...                       [0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [0, 1, 2, 0, 1, 2, 0, 0, 2]]
... )
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3],
...               index=midx)
>>> s
a  lama    speed    45.0
   weight   200.0
   length   1.2
   cow     speed    30.0
   weight   250.0
   length   1.5
b  falcon  speed    320.0
   speed     1.0
   length    0.3
dtype: float64

```

```

>>> s.pop(('a', 'lama'))
speed    45.0
weight   200.0
length   1.2
dtype: float64

```

```
>>> s
a  cow      speed      30.0
      weight    250.0
      length     1.5
b  falcon   speed     320.0
      speed      1.0
      length     0.3
dtype: float64
```

```
>>> s.pop(('b', 'falcon', 'speed'))
(b, falcon, speed)    320.0
(b, falcon, speed)      1.0
dtype: float64
```

### **databricks.koalas.Series.items**

`Series.items()` → Iterable

This is an alias of `iteritems`.

### **databricks.koalas.Series.iteritems**

`Series.iteritems()` → Iterable

Lazily iterate over (index, value) tuples.

This method returns an iterable tuple (index, value). This is convenient if you want to create a lazy iterator.

---

**Note:** Unlike pandas', the `iteritems` in Koalas returns generator rather zip object

---

#### **Returns**

**iterable** Iterable of tuples containing the (index, value) pairs from a Series.

#### **See also:**

**`DataFrame.items`** Iterate over (column name, Series) pairs.

**`DataFrame.iterrows`** Iterate over DataFrame rows as (index, Series) pairs.

### **Examples**

```
>>> s = ks.Series(['A', 'B', 'C'])
>>> for index, value in s.items():
...     print("Index : {}, Value : {}".format(index, value))
Index : 0, Value : A
Index : 1, Value : B
Index : 2, Value : C
```

**databricks.koalas.Series.item**

`Series.item()` → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None]  
 Return the first element of the underlying data as a Python scalar.

**Returns**

**scalar** The first element of Series.

**Raises**

**ValueError** If the data is not length-1.

**Examples**

```
>>> kser = ks.Series([10])
>>> kser.item()
10
```

**databricks.koalas.Series.xs**

`Series.xs(key, level=None)` → databricks.koalas.series.Series

Return cross-section from the Series.

This method takes a *key* argument to select data at a particular level of a MultiIndex.

**Parameters**

**key** [label or tuple of label] Label contained in the index, or partially in a MultiIndex.

**level** [object, defaults to first n levels (n=1 or len(key))] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

**Returns**

**Series** Cross-section from the original Series corresponding to the selected index levels.

**Examples**

```
>>> midx = pd.MultiIndex([[ 'a', 'b', 'c'],
...                       [ 'lama', 'cow', 'falcon'],
...                       [ 'speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3],
...               index=midx)
>>> s
a  lama    speed    45.0
   lama    weight   200.0
   lama    length    1.2
b  cow     speed    30.0
   cow     weight   250.0
   cow     length    1.5
c  falcon  speed    320.0
   falcon  weight    1.0
```

(continues on next page)

(continued from previous page)

```

        length      0.3
dtype: float64

```

**Get values at specified index**

```

>>> s.xs('a')
lama  speed      45.0
      weight     200.0
      length      1.2
dtype: float64

```

**Get values at several indexes**

```

>>> s.xs(('a', 'lama'))
speed      45.0
weight     200.0
length      1.2
dtype: float64

```

**Get values at specified index and level**

```

>>> s.xs('lama', level=1)
a  speed      45.0
   weight     200.0
   length      1.2
dtype: float64

```

**databricks.koalas.Series.get**

`Series.get(key, default=None) → Any`

Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

**Parameters**

**key** [object]

**Returns**

**value** [same type as items contained in object]

**Examples**

```

>>> df = ks.DataFrame({'x':range(3), 'y':['a','b','b'], 'z':['a','b','b']},
...                    columns=['x', 'y', 'z'], index=[10, 20, 20])
>>> df
   x  y  z
10  0  a  a
20  1  b  b
20  2  b  b

```

```

>>> df.get('x')
10    0
20    1
20    2
Name: x, dtype: int64

```

```
>>> df.get(['x', 'y'])
   x  y
10  0  a
20  1  b
20  2  b
```

```
>>> df.x.get(10)
0
```

```
>>> df.x.get(20)
20    1
20    2
Name: x, dtype: int64
```

```
>>> df.x.get(15, -1)
-1
```

### 3.3.5 Binary operator functions

<i>Series.add</i> (other)	Return Addition of series and other, element-wise (binary operator +).
<i>Series.div</i> (other)	Return Floating division of series and other, element-wise (binary operator /).
<i>Series.mul</i> (other)	Return Multiplication of series and other, element-wise (binary operator *).
<i>Series.radd</i> (other)	Return Reverse Addition of series and other, element-wise (binary operator +).
<i>Series.rdiv</i> (other)	Return Reverse Floating division of series and other, element-wise (binary operator /).
<i>Series.rmul</i> (other)	Return Reverse Multiplication of series and other, element-wise (binary operator *).
<i>Series.rsub</i> (other)	Return Reverse Subtraction of series and other, element-wise (binary operator -).
<i>Series.rtruediv</i> (other)	Return Reverse Floating division of series and other, element-wise (binary operator /).
<i>Series.sub</i> (other)	Return Subtraction of series and other, element-wise (binary operator -).
<i>Series.truediv</i> (other)	Return Floating division of series and other, element-wise (binary operator /).
<i>Series.pow</i> (other)	Return Exponential power of series of series and other, element-wise (binary operator **).
<i>Series.rpow</i> (other)	Return Reverse Exponential power of series and other, element-wise (binary operator **).
<i>Series.mod</i> (other)	Return Modulo of series and other, element-wise (binary operator %).
<i>Series.rmod</i> (other)	Return Reverse Modulo of series and other, element-wise (binary operator %).
<i>Series.floordiv</i> (other)	Return Integer division of series and other, element-wise (binary operator //).

continues on next page

Table 23 – continued from previous page

<code>Series.rfloordiv(other)</code>	Return Reverse Integer division of series and other, element-wise (binary operator <code>//</code> ).
<code>Series.divmod(other)</code>	Return Integer division and modulo of series and other, element-wise (binary operator <code>divmod</code> ).
<code>Series.rdivmod(other)</code>	Return Integer division and modulo of series and other, element-wise (binary operator <code>rdivmod</code> ).
<code>Series.combine_first(other)</code>	Combine Series values, choosing the calling Series's values first.
<code>Series.lt(other)</code>	Compare if the current value is less than the other.
<code>Series.gt(other)</code>	Compare if the current value is greater than the other.
<code>Series.le(other)</code>	Compare if the current value is less than or equal to the other.
<code>Series.ge(other)</code>	Compare if the current value is greater than or equal to the other.
<code>Series.ne(other)</code>	Compare if the current value is not equal to the other.
<code>Series.eq(other)</code>	Compare if the current value is equal to the other.
<code>Series.product([axis, numeric_only, min_count])</code>	Return the product of the values.
<code>Series.dot(other)</code>	Compute the dot product between the Series and the columns of other.

**databricks.koalas.Series.add**

`Series.add(other) → databricks.koalas.series.Series`

Return Addition of series and other, element-wise (binary operator `+`).

Equivalent to `series + other`

**Parameters**

**other** [Series or scalar value]

**Returns**

**Series** The result of the operation.

See also:

`Series.radd`

**Examples**

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0 NaN
c  4.0  2.0
d  NaN NaN
```

```
>>> df.a.add(df.b)
a    4.0
```

(continues on next page)

(continued from previous page)

```
b    NaN
c    6.0
d    NaN
dtype: float64
```

```
>>> df.a.radd(df.b)
a    4.0
b    NaN
c    6.0
d    NaN
dtype: float64
```

### `databricks.koalas.Series.div`

`Series.div` (*other*) → `databricks.koalas.series.Series`

Return Floating division of series and other, element-wise (binary operator /).

Equivalent to `series / other`

#### Parameters

**other** [Series or scalar value]

#### Returns

**Series** The result of the operation.

**See also:**

[`Series.rdiv`](#)

### Examples

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0  NaN
c  4.0  2.0
d  NaN  NaN
```

```
>>> df.a.divide(df.b)
a    1.0
b    NaN
c    2.0
d    NaN
dtype: float64
```

```
>>> df.a.rdiv(df.b)
a    1.0
b    NaN
c    0.5
```

(continues on next page)

(continued from previous page)

```
d      NaN
dtype: float64
```

**databricks.koalas.Series.mul**

`Series.mul` (*other*) → `databricks.koalas.series.Series`

Return Multiplication of series and other, element-wise (binary operator \*).

Equivalent to `series * other`

**Parameters**

**other** [Series or scalar value]

**Returns**

**Series** The result of the operation.

**See also:**

[\*Series.rmul\*](#)

**Examples**

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0 NaN
c  4.0  2.0
d  NaN NaN
```

```
>>> df.a.multiply(df.b)
a    4.0
b    NaN
c    8.0
d    NaN
dtype: float64
```

```
>>> df.a.rmul(df.b)
a    4.0
b    NaN
c    8.0
d    NaN
dtype: float64
```



**databricks.koalas.Series.radd**

`Series.radd(other)` → `databricks.koalas.series.Series`

Return Reverse Addition of series and other, element-wise (binary operator +).

Equivalent to `other + series`

**Parameters**

**other** [Series or scalar value]

**Returns**

**Series** The result of the operation.

**See also:**

[\*Series.add\*](#)

**Examples**

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0  NaN
c  4.0  2.0
d  NaN  NaN
```

```
>>> df.a.add(df.b)
a    4.0
b    NaN
c    6.0
d    NaN
dtype: float64
```

```
>>> df.a.radd(df.b)
a    4.0
b    NaN
c    6.0
d    NaN
dtype: float64
```

**databricks.koalas.Series.rdiv**

`Series.rdiv(other)` → `databricks.koalas.series.Series`

Return Reverse Floating division of series and other, element-wise (binary operator /).

Equivalent to `other / series`

**Parameters**

**other** [Series or scalar value]

**Returns**

**Series** The result of the operation.

**See also:**

*[Series.div](#)*

### Examples

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0 NaN
c  4.0  2.0
d  NaN NaN
```

```
>>> df.a.divide(df.b)
a    1.0
b    NaN
c    2.0
d    NaN
dtype: float64
```

```
>>> df.a.rdiv(df.b)
a    1.0
b    NaN
c    0.5
d    NaN
dtype: float64
```

### **databricks.koalas.Series.rmul**

`Series.rmul(other) → databricks.koalas.series.Series`

Return Reverse Multiplication of series and other, element-wise (binary operator \*).

Equivalent to `other * series`

#### **Parameters**

**other** [Series or scalar value]

#### **Returns**

**Series** The result of the operation.

**See also:**

*[Series.mul](#)*

## Examples

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0 NaN
c  4.0  2.0
d  NaN NaN
```

```
>>> df.a.multiply(df.b)
a    4.0
b    NaN
c    8.0
d    NaN
dtype: float64
```

```
>>> df.a.rmul(df.b)
a    4.0
b    NaN
c    8.0
d    NaN
dtype: float64
```

## databricks.koalas.Series.rsub

`Series.rsub(other) → databricks.koalas.series.Series`

Return Reverse Subtraction of series and other, element-wise (binary operator -).

Equivalent to `other - series`

### Parameters

**other** [Series or scalar value]

### Returns

**Series** The result of the operation.

See also:

[\*Series.sub\*](#)

## Examples

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0 NaN
c  4.0  2.0
d  NaN NaN
```

```
>>> df.a.subtract(df.b)
a      0.0
b      NaN
c      2.0
d      NaN
dtype: float64
```

```
>>> df.a.rsub(df.b)
a      0.0
b      NaN
c     -2.0
d      NaN
dtype: float64
```

### **databricks.koalas.Series.rtruediv**

`Series.rtruediv` (*other*) → `databricks.koalas.series.Series`

Return Reverse Floating division of series and other, element-wise (binary operator /).

Equivalent to `other / series`

#### **Parameters**

**other** [Series or scalar value]

#### **Returns**

**Series** The result of the operation.

**See also:**

[\*Series.truediv\*](#)

### **Examples**

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a  b
a  2.0 2.0
b  2.0 NaN
c  4.0 2.0
d  NaN NaN
```

```
>>> df.a.divide(df.b)
a      1.0
b      NaN
c      2.0
d      NaN
dtype: float64
```

```
>>> df.a.rdiv(df.b)
a      1.0
b      NaN
```

(continues on next page)

(continued from previous page)

```
c    0.5
d    NaN
dtype: float64
```

**databricks.koalas.Series.sub**

`Series.sub(other)` → `databricks.koalas.series.Series`

Return Subtraction of series and other, element-wise (binary operator -).

Equivalent to `series - other`

**Parameters**

**other** [Series or scalar value]

**Returns**

**Series** The result of the operation.

**See also:**

[\*Series.rsub\*](#)

**Examples**

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0  NaN
c  4.0  2.0
d  NaN  NaN
```

```
>>> df.a.subtract(df.b)
a    0.0
b    NaN
c    2.0
d    NaN
dtype: float64
```

```
>>> df.a.rsub(df.b)
a    0.0
b    NaN
c   -2.0
d    NaN
dtype: float64
```

**databricks.koalas.Series.truediv**

`Series.truediv(other)` → `databricks.koalas.series.Series`

Return Floating division of series and other, element-wise (binary operator /).

Equivalent to `series / other`

**Parameters**

**other** [Series or scalar value]

**Returns**

**Series** The result of the operation.

**See also:**

[\*Series.rtruediv\*](#)

**Examples**

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0 NaN
c  4.0  2.0
d  NaN NaN
```

```
>>> df.a.divide(df.b)
a    1.0
b    NaN
c    2.0
d    NaN
dtype: float64
```

```
>>> df.a.rdiv(df.b)
a    1.0
b    NaN
c    0.5
d    NaN
dtype: float64
```

**databricks.koalas.Series.pow**

`Series.pow(other)` → `databricks.koalas.series.Series`

Return Exponential power of series of series and other, element-wise (binary operator \*\*).

Equivalent to `series ** other`

**Parameters**

**other** [Series or scalar value]

**Returns**

**Series** The result of the operation.

**See also:**

*Series.rpow*

### Examples

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0 NaN
c  4.0  2.0
d  NaN NaN
```

```
>>> df.a.pow(df.b)
a    4.0
b    NaN
c   16.0
d    NaN
dtype: float64
```

```
>>> df.a.rpow(df.b)
a    4.0
b    NaN
c   16.0
d    NaN
dtype: float64
```

### `databricks.koalas.Series.rpow`

`Series.rpow(other) → databricks.koalas.series.Series`

Return Reverse Exponential power of series and other, element-wise (binary operator `**`).

Equivalent to `other ** series`

#### Parameters

**other** [Series or scalar value]

#### Returns

**Series** The result of the operation.

**See also:**

*Series.pow*

## Examples

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0 NaN
c  4.0  2.0
d  NaN NaN
```

```
>>> df.a.pow(df.b)
a    4.0
b    NaN
c   16.0
d    NaN
dtype: float64
```

```
>>> df.a.rpow(df.b)
a    4.0
b    NaN
c   16.0
d    NaN
dtype: float64
```

## `databricks.koalas.Series.mod`

`Series.mod(other)` → `databricks.koalas.series.Series`

Return Modulo of series and other, element-wise (binary operator %).

Equivalent to `series % other`

### Parameters

**other** [Series or scalar value]

### Returns

**Series** The result of the operation.

See also:

[`Series.rmod`](#)

## Examples

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0 NaN
c  4.0  2.0
d  NaN NaN
```



```
>>> df.a.mod(df.b)
a    0.0
b    NaN
c    0.0
d    NaN
dtype: float64
```

```
>>> df.a.rmod(df.b)
a    0.0
b    NaN
c    2.0
d    NaN
dtype: float64
```

### databricks.koalas.Series.rmod

`Series.rmod(other)` → `databricks.koalas.series.Series`

Return Reverse Modulo of series and other, element-wise (binary operator %).

Equivalent to `other % series`

#### Parameters

**other** [Series or scalar value]

#### Returns

**Series** The result of the operation.

See also:

[`Series.mod`](#)

### Examples

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a  b
a  2.0 2.0
b  2.0 NaN
c  4.0 2.0
d  NaN NaN
```

```
>>> df.a.mod(df.b)
a    0.0
b    NaN
c    0.0
d    NaN
dtype: float64
```

```
>>> df.a.rmod(df.b)
a    0.0
b    NaN
```

(continues on next page)

(continued from previous page)

```
c      2.0
d      NaN
dtype: float64
```

**databricks.koalas.Series.floordiv**

`Series.floordiv` (*other*) → `databricks.koalas.series.Series`

Return Integer division of series and other, element-wise (binary operator `//`).

Equivalent to `series // other`

**Parameters**

**other** [Series or scalar value]

**Returns**

**Series** The result of the operation.

**See also:**

*[Series.rfloordiv](#)*

**Examples**

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a    b
a  2.0  2.0
b  2.0  NaN
c  4.0  2.0
d  NaN  NaN
```

```
>>> df.a.floordiv(df.b)
a    1.0
b    NaN
c    2.0
d    NaN
dtype: float64
```

```
>>> df.a.rfloordiv(df.b)
a    1.0
b    NaN
c    0.0
d    NaN
dtype: float64
```

**databricks.koalas.Series.rfloordiv**

`Series.rfloordiv(other) → databricks.koalas.series.Series`

Return Reverse Integer division of series and other, element-wise (binary operator `//`).

Equivalent to `other // series`

**Parameters**

**other** [Series or scalar value]

**Returns**

**Series** The result of the operation.

**See also:**

[\*Series.floordiv\*](#)

**Examples**

```
>>> df = ks.DataFrame({'a': [2, 2, 4, np.nan],
...                    'b': [2, np.nan, 2, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
>>> df
   a  b
a  2.0 2.0
b  2.0 NaN
c  4.0 2.0
d  NaN NaN
```

```
>>> df.a.floordiv(df.b)
a    1.0
b    NaN
c    2.0
d    NaN
dtype: float64
```

```
>>> df.a.rfloordiv(df.b)
a    1.0
b    NaN
c    0.0
d    NaN
dtype: float64
```

**databricks.koalas.Series.divmod**

`Series.divmod(other) → Tuple[databricks.koalas.series.Series, databricks.koalas.series.Series]`

Return Integer division and modulo of series and other, element-wise (binary operator `divmod`).

**Parameters**

**other** [Series or scalar value]

**Returns**

**2-Tuple of Series** The result of the operation.

See also:

*Series.rdivmod*

### **databricks.koalas.Series.rdivmod**

`Series.rdivmod(other) → Tuple[databricks.koalas.series.Series, databricks.koalas.series.Series]`  
Return Integer division and modulo of series and other, element-wise (binary operator *rdivmod*).

#### **Parameters**

**other** [Series or scalar value]

#### **Returns**

**2-Tuple of Series** The result of the operation.

See also:

*Series.divmod*

### **databricks.koalas.Series.combine\_first**

`Series.combine_first(other) → databricks.koalas.series.Series`  
Combine Series values, choosing the calling Series's values first.

#### **Parameters**

**other** [Series] The value(s) to be combined with the *Series*.

#### **Returns**

**Series** The result of combining the Series with the other object.

See also:

**Series.combine** Perform elementwise operation on two Series using a given function.

### **Notes**

Result index will be the union of the two indexes.

### **Examples**

```
>>> s1 = ks.Series([1, np.nan])
>>> s2 = ks.Series([3, 4])
>>> with ks.option_context("compute.ops_on_diff_frames", True):
...     s1.combine_first(s2)
0    1.0
1    4.0
dtype: float64
```

**databricks.koalas.Series.lt**

`Series.lt(other)` → `databricks.koalas.series.Series`

Compare if the current value is less than the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                    'b': [1, np.nan, 1, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.a < 1
a    False
b    False
c    False
d    False
Name: a, dtype: bool
```

```
>>> df.b.lt(2)
a     True
b    False
c     True
d    False
Name: b, dtype: bool
```

**databricks.koalas.Series.gt**

`Series.gt(other)` → `databricks.koalas.series.Series`

Compare if the current value is greater than the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                    'b': [1, np.nan, 1, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.a > 1
a    False
b     True
c     True
d     True
Name: a, dtype: bool
```

```
>>> df.b.gt(1)
a    False
b    False
c    False
d    False
Name: b, dtype: bool
```

**databricks.koalas.Series.le**

`Series.le(other)` → `databricks.koalas.series.Series`

Compare if the current value is less than or equal to the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                     'b': [1, np.nan, 1, np.nan]},
...                     index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.a <= 2
a      True
b      True
c     False
d     False
Name: a, dtype: bool
```

```
>>> df.b.le(2)
a      True
b     False
c      True
d     False
Name: b, dtype: bool
```

**databricks.koalas.Series.ge**

`Series.ge(other)` → `databricks.koalas.series.Series`

Compare if the current value is greater than or equal to the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                     'b': [1, np.nan, 1, np.nan]},
...                     index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.a >= 2
a     False
b      True
c      True
d      True
Name: a, dtype: bool
```

```
>>> df.b.ge(2)
a     False
b     False
c     False
d     False
Name: b, dtype: bool
```

**databricks.koalas.Series.ne**

`Series.ne(other) → databricks.koalas.series.Series`

Compare if the current value is not equal to the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                    'b': [1, np.nan, 1, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.a != 1
a    False
b     True
c     True
d     True
Name: a, dtype: bool
```

```
>>> df.b.ne(1)
a    False
b     True
c    False
d     True
Name: b, dtype: bool
```

**databricks.koalas.Series.eq**

`Series.eq(other) → bool`

Compare if the current value is equal to the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                    'b': [1, np.nan, 1, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.a == 1
a     True
b    False
c    False
d    False
Name: a, dtype: bool
```

```
>>> df.b.eq(1)
a     True
b    False
c     True
d    False
Name: b, dtype: bool
```

**databricks.koalas.Series.product**

`Series.product` (*axis*: `Union[int, str] = None`, *numeric\_only*: `bool = None`, *min\_count*: `int = 0`)  $\rightarrow$  `Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]`

Return the product of the values.

**Note:** unlike pandas', Koalas' emulates product by `exp(sum(log(...)))` trick. Therefore, it only works for positive numbers.

**Parameters**

**axis** `[[index (0), columns (1)]]` Axis for the function to be applied on.

**numeric\_only** `[bool, default None]` Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**min\_count** `[int, default 0]` The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

**Examples**

On a DataFrame:

Non-numeric type column is not included to the result.

```
>>> kdf = ks.DataFrame({'A': [1, 2, 3, 4, 5],
...                     'B': [10, 20, 30, 40, 50],
...                     'C': ['a', 'b', 'c', 'd', 'e']})
>>> kdf
   A  B  C
0  1 10  a
1  2 20  b
2  3 30  c
3  4 40  d
4  5 50  e
```

```
>>> kdf.prod()
A      120
B    12000000
dtype: int64
```

If there is no numeric type columns, returns empty Series.

```
>>> ks.DataFrame({"key": ['a', 'b', 'c'], "val": ['x', 'y', 'z']}).prod()
Series([], dtype: float64)
```

On a Series:

```
>>> ks.Series([1, 2, 3, 4, 5]).prod()
120
```

By default, the product of an empty or all-NA Series is 1

```
>>> ks.Series([]).prod()
1.0
```



This can be controlled with the `min_count` parameter

```
>>> ks.Series([]).prod(min_count=1)
nan
```

### **databricks.koalas.Series.dot**

`Series.dot (other: Union[Series, databricks.koalas.frame.DataFrame]) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]`  
 Compute the dot product between the Series and the columns of other.

This method computes the dot product between the Series and another one, or the Series and each columns of a DataFrame.

It can also be called using `self @ other` in Python `>= 3.5`.

**Note:** This API is slightly different from pandas when indexes from both Series are not aligned. To match with pandas', it requires to read the whole data for, for example, counting. pandas raises an exception; however, Koalas just proceeds and performs by ignoring mismatches with NaN permissively.

```
>>> pdf1 = pd.Series([1, 2, 3], index=[0, 1, 2])
>>> pdf2 = pd.Series([1, 2, 3], index=[0, 1, 3])
>>> pdf1.dot(pdf2)
...
ValueError: matrices are not aligned
```

```
>>> kdf1 = ks.Series([1, 2, 3], index=[0, 1, 2])
>>> kdf2 = ks.Series([1, 2, 3], index=[0, 1, 3])
>>> kdf1.dot(kdf2)
5
```

#### **Parameters**

**other** [Series, DataFrame.] The other object to compute the dot product with its columns.

#### **Returns**

**scalar, Series** Return the dot product of the Series and other if other is a Series, the Series of the dot product of Series and each rows of other if other is a DataFrame.

#### **Notes**

The Series and other has to share the same index if other is a Series or a DataFrame.

## Examples

```
>>> s = ks.Series([0, 1, 2, 3])
```

```
>>> s.dot(s)
14
```

```
>>> s @ s
14
```

```
>>> kdf = ks.DataFrame({'x': [0, 1, 2, 3], 'y': [0, -1, -2, -3]})
>>> kdf
   x  y
0  0  0
1  1 -1
2  2 -2
3  3 -3
```

```
>>> with ks.option_context("compute.ops_on_diff_frames", True):
...     s.dot(kdf)
...
x      14
y     -14
dtype: int64
```

### 3.3.6 Function application, GroupBy & Window

<code>Series.apply(func[, args])</code>	Invoke function on values of Series.
<code>Series.agg(func)</code>	Aggregate using one or more operations over the specified axis.
<code>Series.aggregate(func)</code>	Aggregate using one or more operations over the specified axis.
<code>Series.transform(func[, axis])</code>	Call <code>func</code> producing the same type as <i>self</i> with transformed values and that has the same axis length as input.
<code>Series.map(arg)</code>	Map values of Series according to input correspondence.
<code>Series.groupby(by[, axis, as_index, dropna])</code>	Group DataFrame or Series using a Series of columns.
<code>Series.rolling(window[, min_periods])</code>	Provide rolling transformations.
<code>Series.expanding([min_periods])</code>	Provide expanding transformations.
<code>Series.pipe(func, *args, **kwargs)</code>	Apply <code>func(self, *args, **kwargs)</code> .

**databricks.koalas.Series.apply**

`Series.apply(func, args=(), **kws) → databricks.koalas.series.Series`

Invoke function on values of Series.

Can be a Python function that only works on the Series.

**Note:** this API executes the function once to infer the type which is potentially expensive, for instance, when the dataset is created after aggregations or sorting.

To avoid this, specify return type in `func`, for instance, as below:

```
>>> def square(x) -> np.int32:
...     return x ** 2
```

Koalas uses return type hint and does not try to infer the type.

**Parameters**

**func** [function] Python function to apply. Note that type hint for return type is required.

**args** [tuple] Positional arguments passed to `func` after the series value.

**\*\*kws** Additional keyword arguments passed to `func`.

**Returns**

**Series**

See also:

**`Series.aggregate`** Only perform aggregating type operations.

**`Series.transform`** Only perform transforming type operations.

**`DataFrame.apply`** The equivalent function for `DataFrame`.

**Examples**

Create a Series with typical summer temperatures for each city.

```
>>> s = ks.Series([20, 21, 12],
...                 index=['London', 'New York', 'Helsinki'])
>>> s
London      20
New York    21
Helsinki    12
dtype: int64
```

Square the values by defining a function and passing it as an argument to `apply()`.

```
>>> def square(x) -> np.int64:
...     return x ** 2
>>> s.apply(square)
London      400
New York    441
Helsinki    144
dtype: int64
```

Define a custom function that needs additional positional arguments and pass these additional arguments using the `args` keyword

```
>>> def subtract_custom_value(x, custom_value) -> np.int64:
...     return x - custom_value
```

```
>>> s.apply(subtract_custom_value, args=(5,))
London      15
New York    16
Helsinki     7
dtype: int64
```

Define a custom function that takes keyword arguments and pass these arguments to `apply`

```
>>> def add_custom_values(x, **kwargs) -> np.int64:
...     for month in kwargs:
...         x += kwargs[month]
...     return x
```

```
>>> s.apply(add_custom_values, june=30, july=20, august=25)
London      95
New York    96
Helsinki    87
dtype: int64
```

Use a function from the Numpy library

```
>>> def numpy_log(col) -> np.float64:
...     return np.log(col)
>>> s.apply(numpy_log)
London      2.995732
New York    3.044522
Helsinki    2.484907
dtype: float64
```

You can omit the type hint and let Koalas infer its type.

```
>>> s.apply(np.log)
London      2.995732
New York    3.044522
Helsinki    2.484907
dtype: float64
```

## databricks.koalas.Series.agg

`Series.agg(func: Union[str, List[str]]) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]`

Aggregate using one or more operations over the specified axis.

### Parameters

**func** [str or a list of str] function name(s) as string apply to series.

### Returns

**scalar, Series** The return can be: - scalar : when `Series.agg` is called with single function - Series : when `Series.agg` is called with several functions

**See also:**

**`Series.apply`** Invoke function on a Series.

**`Series.transform`** Only perform transforming type operations.

**`Series.groupby`** Perform operations over groups.

**`DataFrame.aggregate`** The equivalent function for DataFrame.

**Notes**

`agg` is an alias for `aggregate`. Use the alias.

**Examples**

```
>>> s = ks.Series([1, 2, 3, 4])
>>> s.agg('min')
1
```

```
>>> s.agg(['min', 'max']).sort_index()
max      4
min      1
dtype: int64
```

**databricks.koalas.Series.aggregate**

**`Series.aggregate`** (*func: Union[str, List[str]]*) → Union[int, float, str, bytes, decimal.Decimal, date-time.date, None, databricks.koalas.series.Series]

Aggregate using one or more operations over the specified axis.

**Parameters**

**func** [str or a list of str] function name(s) as string apply to series.

**Returns**

**scalar, Series** The return can be: - scalar : when `Series.agg` is called with single function - Series : when `Series.agg` is called with several functions

**See also:**

**`Series.apply`** Invoke function on a Series.

**`Series.transform`** Only perform transforming type operations.

**`Series.groupby`** Perform operations over groups.

**`DataFrame.aggregate`** The equivalent function for DataFrame.

## Notes

*agg* is an alias for *aggregate*. Use the alias.

## Examples

```
>>> s = ks.Series([1, 2, 3, 4])
>>> s.agg('min')
1
```

```
>>> s.agg(['min', 'max']).sort_index()
max      4
min      1
dtype: int64
```

## databricks.koalas.Series.transform

`Series.transform(func, axis=0, *args, **kwargs) → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]`  
Call *func* producing the same type as *self* with transformed values and that has the same axis length as input.

---

**Note:** this API executes the function once to infer the type which is potentially expensive, for instance, when the dataset is created after aggregations or sorting.

To avoid this, specify return type in *func*, for instance, as below:

```
>>> def square(x) -> np.int32:
...     return x ** 2
```

Koalas uses return type hint and does not try to infer the type.

---

### Parameters

**func** [function or list] A function or a list of functions to use for transforming the data.

**axis** [int, default 0 or 'index'] Can only be set to 0 at the moment.

**\*args** Positional arguments to pass to *func*.

**\*\*kwargs** Keyword arguments to pass to *func*.

### Returns

An instance of the same type with *self* that must have the same length as input.

See also:

**Series.aggregate** Only perform aggregating type operations.

**Series.apply** Invoke function on Series.

**DataFrame.transform** The equivalent function for DataFrame.

## Examples

```
>>> s = ks.Series(range(3))
>>> s
0    0
1    1
2    2
dtype: int64
```

```
>>> def sqrt(x) -> float:
...     return np.sqrt(x)
>>> s.transform(sqrt)
0    0.000000
1    1.000000
2    1.414214
dtype: float64
```

Even though the resulting instance must have the same length as the input, it is possible to provide several input functions:

```
>>> def exp(x) -> float:
...     return np.exp(x)
>>> s.transform([sqrt, exp])
      sqrt      exp
0  0.000000  1.000000
1  1.000000  2.718282
2  1.414214  7.389056
```

You can omit the type hint and let Koalas infer its type.

```
>>> s.transform([np.sqrt, np.exp])
      sqrt      exp
0  0.000000  1.000000
1  1.000000  2.718282
2  1.414214  7.389056
```

## databricks.koalas.Series.map

`Series.map(arg) → databricks.koalas.series.Series`

Map values of Series according to input correspondence.

Used for substituting each value in a Series with another value, that may be derived from a function, a dict.

**Note:** make sure the size of the dictionary is not huge because it could downgrade the performance or throw `OutOfMemoryError` due to a huge expression within Spark. Consider the input as a functions as an alternative instead in this case.

### Parameters

**arg** [function or dict] Mapping correspondence.

### Returns

**Series** Same index as caller.

See also:

**`Series.apply`** For applying more complex functions on a Series.

**`DataFrame.applymap`** Apply a function elementwise on a whole DataFrame.

## Notes

When `arg` is a dictionary, values in Series that are not in the dictionary (as keys) are converted to `None`. However, if the dictionary is a `dict` subclass that defines `__missing__` (i.e. provides a method for default values), then this default is used rather than `None`.

## Examples

```
>>> s = ks.Series(['cat', 'dog', None, 'rabbit'])
>>> s
0      cat
1      dog
2      None
3    rabbit
dtype: object
```

`map` accepts a `dict`. Values that are not found in the `dict` are converted to `None`, unless the `dict` has a default value (e.g. `defaultdict`):

```
>>> s.map({'cat': 'kitten', 'dog': 'puppy'})
0      kitten
1      puppy
2      None
3      None
dtype: object
```

It also accepts a function:

```
>>> def format(x) -> str:
...     return 'I am a {}'.format(x)
```

```
>>> s.map(format)
0      I am a cat
1      I am a dog
2      I am a None
3      I am a rabbit
dtype: object
```



**databricks.koalas.Series.groupby**

`Series.groupby` (*by*, *axis=0*, *as\_index: bool = True*, *dropna: bool = True*) → Union[DataFrameGroupBy, SeriesGroupBy]  
 Group DataFrame or Series using a Series of columns.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

**Parameters**

**by** [Series, label, or list of labels] Used to determine the groups for the groupby. If Series is passed, the Series or dict VALUES will be used to determine the groups. A label or list of labels may be passed to group by the columns in `self`.

**axis** [int, default 0 or 'index'] Can only be set to 0 at the moment.

**as\_index** [bool, default True] For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. `as_index=False` is effectively “SQL-style” grouped output.

**dropna** [bool, default True] If True, and if group keys contain NA values, NA values together with row/column will be dropped. If False, NA values will also be treated as the key in groups.

**Returns**

**DataFrameGroupBy or SeriesGroupBy** Depends on the calling object and returns groupby object that contains information about the groups.

See also:

`koalas.groupby.GroupBy`

**Examples**

```
>>> df = ks.DataFrame({'Animal': ['Falcon', 'Falcon',
...                               'Parrot', 'Parrot'],
...                    'Max Speed': [380., 370., 24., 26.]},
...                    columns=['Animal', 'Max Speed'])
>>> df
   Animal  Max Speed
0  Falcon    380.0
1  Falcon    370.0
2  Parrot     24.0
3  Parrot     26.0
```

```
>>> df.groupby(['Animal']).mean().sort_index()
   Animal
Falcon    375.0
Parrot     25.0
```

```
>>> df.groupby(['Animal'], as_index=False).mean().sort_values('Animal')
...
   Animal  Max Speed
...Falcon    375.0
...Parrot     25.0
```

We can also choose to include NA in group keys or not by setting dropna parameter, the default setting is True:

```
>>> l = [[1, 2, 3], [1, None, 4], [2, 1, 3], [1, 2, 2]]
>>> df = ks.DataFrame(l, columns=["a", "b", "c"])
>>> df.groupby(by=["b"]).sum().sort_index()
      a  c
b
1.0  2  3
2.0  2  5
```

```
>>> df.groupby(by=["b"], dropna=False).sum().sort_index()
      a  c
b
1.0  2  3
2.0  2  5
NaN  1  4
```

### **databricks.koalas.Series.rolling**

`Series.rolling(window, min_periods=None)` → `databricks.koalas.window.Rolling`  
Provide rolling transformations.

---

**Note:** ‘min\_periods’ in Koalas works as a fixed window size unlike pandas. Unlike pandas, NA is also counted as the period. This might be changed in the near future.

---

#### **Parameters**

**window** [int, or offset] Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

**min\_periods** [int, default None] Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, min\_periods will default to 1. Otherwise, min\_periods will default to the size of the window.

#### **Returns**

**a Window sub-classed for the particular operation**

### **databricks.koalas.Series.expanding**

`Series.expanding(min_periods=1)` → `databricks.koalas.window.Expanding`  
Provide expanding transformations.

---

**Note:** ‘min\_periods’ in Koalas works as a fixed window size unlike pandas. Unlike pandas, NA is also counted as the period. This might be changed in the near future.

---

#### **Parameters**

**min\_periods** [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

#### **Returns**

**a Window sub-classed for the particular operation**

**databricks.koalas.Series.pipe**

`Series.pipe(func, *args, **kwargs) → Any`  
 Apply `func(self, *args, **kwargs)`.

**Parameters**

**func** [function] function to apply to the DataFrame. `args`, and `kwargs` are passed into `func`. Alternatively a (callable, `data_keyword`) tuple where `data_keyword` is a string indicating the keyword of callable that expects the DataFrames.

**args** [iterable, optional] positional arguments passed into `func`.

**kwargs** [mapping, optional] a dictionary of keyword arguments passed into `func`.

**Returns**

**object** [the return type of `func`.]

**Notes**

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. For example, given

```
>>> df = ks.DataFrame({'category': ['A', 'A', 'B'],
...                    'col1': [1, 2, 3],
...                    'col2': [4, 5, 6]},
...                  columns=['category', 'col1', 'col2'])
>>> def keep_category_a(df):
...     return df[df['category'] == 'A']
>>> def add_one(df, column):
...     return df.assign(col3=df[column] + 1)
>>> def multiply(df, column1, column2):
...     return df.assign(col4=df[column1] * df[column2])
```

instead of writing

```
>>> multiply(add_one(keep_category_a(df), column="col1"), column1="col2", column2=
↪ "col3")
   category  col1  col2  col3  col4
0         A     1     4     2     8
1         A     2     5     3    15
```

You can write

```
>>> (df.pipe(keep_category_a)
...   .pipe(add_one, column="col1")
...   .pipe(multiply, column1="col2", column2="col3")
... )
   category  col1  col2  col3  col4
0         A     1     4     2     8
1         A     2     5     3    15
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `df`:

```
>>> def multiply_2(column1, df, column2):
...     return df.assign(col4=df[column1] * df[column2])
```

Then you can write

```
>>> (df.pipe(keep_category_a)
...     .pipe(add_one, column="col1")
...     .pipe((multiply_2, 'df'), column1="col2", column2="col3")
... )
  category  col1  col2  col3  col4
0         A     1     4     2     8
1         A     2     5     3    15
```

You can use lambda as well

```
>>> ks.Series([1, 2, 3]).pipe(lambda x: (x + 1).rename("value"))
0     2
1     3
2     4
Name: value, dtype: int64
```

### 3.3.7 Computations / Descriptive Stats

<code>Series.abs()</code>	Return a Series/DataFrame with absolute numeric value of each element.
<code>Series.all([axis])</code>	Return whether all elements are True.
<code>Series.any([axis])</code>	Return whether any element is True.
<code>Series.between(left, right[, inclusive])</code>	Return boolean Series equivalent to <code>left &lt;= series &lt;= right</code> .
<code>Series.clip([lower, upper])</code>	Trim values at input threshold(s).
<code>Series.corr(other[, method])</code>	Compute correlation with <i>other</i> Series, excluding missing values.
<code>Series.count([axis, numeric_only])</code>	Count non-NA cells for each column.
<code>Series.cummax([skipna])</code>	Return cumulative maximum over a DataFrame or Series axis.
<code>Series.cummin([skipna])</code>	Return cumulative minimum over a DataFrame or Series axis.
<code>Series.cumsum([skipna])</code>	Return cumulative sum over a DataFrame or Series axis.
<code>Series.cumprod([skipna])</code>	Return cumulative product over a DataFrame or Series axis.
<code>Series.describe([percentiles])</code>	Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
<code>Series.filter([items, like, regex, axis])</code>	Subset rows or columns of dataframe according to labels in the specified index.
<code>Series.kurt([axis, numeric_only])</code>	Return unbiased kurtosis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>Series.mad()</code>	Return the mean absolute deviation of values.
<code>Series.max([axis, numeric_only])</code>	Return the maximum of the values.
<code>Series.mean([axis, numeric_only])</code>	Return the mean of the values.
<code>Series.min([axis, numeric_only])</code>	Return the minimum of the values.
<code>Series.mode([dropna])</code>	Return the mode(s) of the dataset.
<code>Series.nlargest([n])</code>	Return the largest <i>n</i> elements.
<code>Series.nsmallest([n])</code>	Return the smallest <i>n</i> elements.

continues on next page

Table 25 – continued from previous page

<code>Series.pct_change([periods])</code>	Percentage change between the current and a prior element.
<code>Series.prod([axis, numeric_only, min_count])</code>	Return the product of the values.
<code>Series.nunique([dropna, approx, rsd])</code>	Return number of unique elements in the object.
<code>Series.is_unique</code>	Return boolean if values in the object are unique
<code>Series.quantile([q, accuracy])</code>	Return value at the given quantile.
<code>Series.rank([method, ascending])</code>	Compute numerical data ranks (1 through n) along axis.
<code>Series.sem([axis, ddof, numeric_only])</code>	Return unbiased standard error of the mean over requested axis.
<code>Series.skew([axis, numeric_only])</code>	Return unbiased skew normalized by N-1.
<code>Series.std([axis, ddof, numeric_only])</code>	Return sample standard deviation.
<code>Series.sum([axis, numeric_only, min_count])</code>	Return the sum of the values.
<code>Series.median([axis, numeric_only, accuracy])</code>	Return the median of the values for the requested axis.
<code>Series.var([axis, ddof, numeric_only])</code>	Return unbiased variance.
<code>Series.kurtosis([axis, numeric_only])</code>	Return unbiased kurtosis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>Series.unique()</code>	Return unique values of Series object.
<code>Series.value_counts([normalize, sort, ...])</code>	Return a Series containing counts of unique values.
<code>Series.round([decimals])</code>	Round each value in a Series to the given number of decimals.
<code>Series.diff([periods])</code>	First discrete difference of element.
<code>Series.is_monotonic</code>	Return boolean if values in the object are monotonically increasing.
<code>Series.is_monotonic_increasing</code>	Return boolean if values in the object are monotonically increasing.
<code>Series.is_monotonic_decreasing</code>	Return boolean if values in the object are monotonically decreasing.

**databricks.koalas.Series.abs**

`Series.abs()` → Union[DataFrame, Series]

Return a Series/DataFrame with absolute numeric value of each element.

**Returns**

**abs** [Series/DataFrame containing the absolute value of each element.]

**Examples**

Absolute numeric values in a Series.

```
>>> s = ks.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a DataFrame.

```
>>> df = ks.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... },
...     columns=['a', 'b', 'c'])
>>> df.abs()
   a  b  c
0  4 10 100
1  5 20  50
2  6 30  30
3  7 40  50
```

### **databricks.koalas.Series.all**

`Series.all (axis: Union[int, str] = 0) → bool`

Return whether all elements are True.

Returns True unless there at least one element within a series that is False or equivalent (e.g. zero or empty)

#### **Parameters**

**axis** [{0 or 'index'}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.

### **Examples**

```
>>> ks.Series([True, True]).all()
True
```

```
>>> ks.Series([True, False]).all()
False
```

```
>>> ks.Series([0, 1]).all()
False
```

```
>>> ks.Series([1, 2, 3]).all()
True
```

```
>>> ks.Series([True, True, None]).all()
True
```

```
>>> ks.Series([True, False, None]).all()
False
```

```
>>> ks.Series([]).all()
True
```

```
>>> ks.Series([np.nan]).all()
True
```

```
>>> df = ks.Series([True, False, None]).rename("a").to_frame()
>>> df.set_index("a").index.all()
False
```

### databricks.koalas.Series.any

`Series.any (axis: Union[int, str] = 0) → bool`

Return whether any element is True.

Returns False unless there at least one element within a series that is True or equivalent (e.g. non-zero or non-empty).

#### Parameters

**axis** [{0 or 'index'}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.

### Examples

```
>>> ks.Series([False, False]).any()
False
```

```
>>> ks.Series([True, False]).any()
True
```

```
>>> ks.Series([0, 0]).any()
False
```

```
>>> ks.Series([0, 1, 2]).any()
True
```

```
>>> ks.Series([False, False, None]).any()
False
```

```
>>> ks.Series([True, False, None]).any()
True
```

```
>>> ks.Series([]).any()
False
```

```
>>> ks.Series([np.nan]).any()
False
```

```
>>> df = ks.Series([True, False, None]).rename("a").to_frame()
>>> df.set_index("a").index.any()
True
```

## `databricks.koalas.Series.between`

`Series.between` (*left*, *right*, *inclusive=True*) → `databricks.koalas.series.Series`

Return boolean Series equivalent to `left <= series <= right`. This function returns a boolean vector containing *True* wherever the corresponding Series element is between the boundary values *left* and *right*. NA values are treated as *False*.

### Parameters

**left** [scalar or list-like] Left boundary.

**right** [scalar or list-like] Right boundary.

**inclusive** [bool, default True] Include boundaries.

### Returns

**Series** Series representing whether each element is between left and right (inclusive).

See also:

[`Series.gt`](#) Greater than of series and other.

[`Series.lt`](#) Less than of series and other.

### Notes

This function is equivalent to `(left <= ser) & (ser <= right)`

### Examples

```
>>> s = ks.Series([2, 0, 4, 8, np.nan])
```

Boundary values are included by default:

```
>>> s.between(1, 4)
0      True
1     False
2      True
3     False
4     False
dtype: bool
```

With *inclusive* set to *False* boundary values are excluded:

```
>>> s.between(1, 4, inclusive=False)
0      True
1     False
2     False
3     False
4     False
dtype: bool
```

*left* and *right* can be any scalar value:



```
>>> s = ks.Series(['Alice', 'Bob', 'Carol', 'Eve'])
>>> s.between('Anna', 'Daniel')
0    False
1     True
2     True
3    False
dtype: bool
```

### **databricks.koalas.Series.clip**

**Series.clip**(*lower*: Union[float, int] = None, *upper*: Union[float, int] = None) → databricks.koalas.series.Series  
Trim values at input threshold(s).

Assigns values outside boundary to boundary values.

#### **Parameters**

**lower** [float or int, default None] Minimum threshold value. All values below this threshold will be set to it.

**upper** [float or int, default None] Maximum threshold value. All values above this threshold will be set to it.

#### **Returns**

**Series** Series with the values outside the clip boundaries replaced

### **Notes**

One difference between this implementation and pandas is that running `pd.Series(['a', 'b']).clip(0, 1)` will crash with “TypeError: ‘<=’ not supported between instances of ‘str’ and ‘int’” while `ks.Series(['a', 'b']).clip(0, 1)` will output the original Series, simply ignoring the incompatible types.

### **Examples**

```
>>> ks.Series([0, 2, 4]).clip(1, 3)
0    1
1    2
2    3
dtype: int64
```

### **databricks.koalas.Series.corr**

**Series.corr**(*other*, *method*='pearson') → float  
Compute correlation with *other* Series, excluding missing values.

#### **Parameters**

**other** [Series]

**method** [{‘pearson’, ‘spearman’}]

- pearson : standard correlation coefficient
- spearman : Spearman rank correlation

**Returns****correlation** [float]**Notes**

There are behavior differences between Koalas and pandas.

- the *method* argument only accepts ‘pearson’, ‘spearman’
- the data should not contain NaNs. Koalas will return an error.
- Koalas doesn’t support the following argument(s).
  - *min\_periods* argument is not supported

**Examples**

```
>>> df = ks.DataFrame({'s1': [.2, .0, .6, .2],  
...                   's2': [.3, .6, .0, .1]})  
>>> s1 = df.s1  
>>> s2 = df.s2  
>>> s1.corr(s2, method='pearson')  
-0.851064...
```

```
>>> s1.corr(s2, method='spearman')  
-0.948683...
```

**databricks.koalas.Series.count**

`Series.count` (*axis*: Union[int, str] = None, *numeric\_only*: bool = False) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]

Count non-NA cells for each column.

The values *None*, *NaN* are considered NA.

**Parameters**

**axis** [{0 or ‘index’, 1 or ‘columns’}, default 0] If 0 or ‘index’ counts are generated for each column. If 1 or ‘columns’ counts are generated for each row.

**numeric\_only** [bool, default False] If True, include only float, int, boolean columns. This parameter is mainly for pandas compatibility.

**Returns**

**max** [scalar for a Series, and a Series for a DataFrame.]

See also:

**DataFrame.shape** Number of DataFrame rows and columns (including NA elements).

**DataFrame.isna** Boolean same-sized DataFrame showing places of NA elements.

## Examples

Constructing DataFrame from a dictionary:

```
>>> df = ks.DataFrame({"Person":
...                     ["John", "Myla", "Lewis", "John", "Myla"],
...                     "Age": [24., np.nan, 21., 33, 26],
...                     "Single": [False, True, True, True, False]},
...                     columns=["Person", "Age", "Single"])
>>> df
   Person  Age  Single
0   John  24.0  False
1   Myla   NaN   True
2  Lewis  21.0   True
3   John  33.0   True
4   Myla  26.0  False
```

Notice the uncounted NA values:

```
>>> df.count()
Person      5
Age         4
Single      5
dtype: int64
```

```
>>> df.count(axis=1)
0      3
1      2
2      3
3      3
4      3
dtype: int64
```

On a Series:

```
>>> df['Person'].count()
5
```

```
>>> df['Age'].count()
4
```

## databricks.koalas.Series.cummax

`Series.cummax(skipna: bool = True) → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]`

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

---

**Note:** the current implementation of cummax uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

## Parameters

**skipna** [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

### Returns

**DataFrame or Series**

See also:

**DataFrame.max** Return the maximum over DataFrame axis.

**DataFrame.cummax** Return cumulative maximum over DataFrame axis.

**DataFrame.cummin** Return cumulative minimum over DataFrame axis.

**DataFrame.cumsum** Return cumulative sum over DataFrame axis.

**DataFrame.cumprod** Return cumulative product over DataFrame axis.

**Series.max** Return the maximum over Series axis.

**Series.cummax** Return cumulative maximum over Series axis.

**Series.cummin** Return cumulative minimum over Series axis.

**Series.cumsum** Return cumulative sum over Series axis.

**Series.cumprod** Return cumulative product over Series axis.

### Examples

```
>>> df = ks.DataFrame([[2.0, 1.0], [3.0, None], [1.0, 0.0]], columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column.

```
>>> df.cummax()
   A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

It works identically in Series.

```
>>> df.B.cummax()
0    1.0
1    NaN
2    1.0
Name: B, dtype: float64
```

**databricks.koalas.Series.cummin**

`Series.cummin` (*skipna*: *bool* = *True*) → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

---

**Note:** the current implementation of cummin uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

**Parameters**

**skipna** [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**Returns**

**DataFrame or Series**

See also:

**DataFrame.min** Return the minimum over DataFrame axis.

**DataFrame.cummax** Return cumulative maximum over DataFrame axis.

**DataFrame.cummin** Return cumulative minimum over DataFrame axis.

**DataFrame.cumsum** Return cumulative sum over DataFrame axis.

**Series.min** Return the minimum over Series axis.

**Series.cummax** Return cumulative maximum over Series axis.

**Series.cummin** Return cumulative minimum over Series axis.

**Series.cumsum** Return cumulative sum over Series axis.

**Series.cumprod** Return cumulative product over Series axis.

**Examples**

```
>>> df = ks.DataFrame([[2.0, 1.0], [3.0, None], [1.0, 0.0]], columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0 NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column.

```
>>> df.cummin()
   A    B
0  2.0  1.0
1  2.0 NaN
2  1.0  0.0
```

It works identically in Series.

```
>>> df.A.cummin()
0    2.0
1    2.0
2    1.0
Name: A, dtype: float64
```

### **databricks.koalas.Series.cumsum**

`Series.cumsum(skipna: bool = True) → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]`

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

---

**Note:** the current implementation of `cumsum` uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

#### **Parameters**

**skipna** [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

#### **Returns**

**DataFrame or Series**

See also:

**DataFrame.sum** Return the sum over DataFrame axis.

**DataFrame.cummax** Return cumulative maximum over DataFrame axis.

**DataFrame.cummin** Return cumulative minimum over DataFrame axis.

**DataFrame.cumsum** Return cumulative sum over DataFrame axis.

**DataFrame.cumprod** Return cumulative product over DataFrame axis.

**Series.sum** Return the sum over Series axis.

**Series.cummax** Return cumulative maximum over Series axis.

**Series.cummin** Return cumulative minimum over Series axis.

**Series.cumsum** Return cumulative sum over Series axis.

**Series.cumprod** Return cumulative product over Series axis.

## Examples

```
>>> df = ks.DataFrame([[2.0, 1.0], [3.0, None], [1.0, 0.0]], columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column.

```
>>> df.cumsum()
   A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

It works identically in Series.

```
>>> df.A.cumsum()
0    2.0
1    5.0
2    6.0
Name: A, dtype: float64
```

## databricks.koalas.Series.cumprod

`Series.cumprod(skipna: bool = True) → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]`

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

---

**Note:** the current implementation of `cumprod` uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---



---

**Note:** unlike pandas', Koalas' emulates cumulative product by `exp(sum(log(...)))` trick. Therefore, it only works for positive numbers.

---

### Parameters

**skipna** [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

### Returns

**DataFrame or Series**

### Raises

**Exception** [If the values is equal to or lower than 0.]

See also:

**DataFrame.cummax** Return cumulative maximum over DataFrame axis.

**DataFrame.cummin** Return cumulative minimum over DataFrame axis.

**DataFrame.cumsum** Return cumulative sum over DataFrame axis.

**DataFrame.cumprod** Return cumulative product over DataFrame axis.

**Series.cummax** Return cumulative maximum over Series axis.

**Series.cummin** Return cumulative minimum over Series axis.

**Series.cumsum** Return cumulative sum over Series axis.

**Series.cumprod** Return cumulative product over Series axis.

## Examples

```
>>> df = ks.DataFrame([[2.0, 1.0], [3.0, None], [4.0, 10.0]], columns=list('AB'))
>>> df
   A      B
0  2.0    1.0
1  3.0    NaN
2  4.0   10.0
```

By default, iterates over rows and finds the sum in each column.

```
>>> df.cumprod()
   A      B
0  2.0    1.0
1  6.0    NaN
2 24.0   10.0
```

It works identically in Series.

```
>>> df.A.cumprod()
0      2.0
1      6.0
2     24.0
Name: A, dtype: float64
```

## databricks.koalas.Series.describe

**Series.describe** (*percentiles: Optional[List[float]] = None*) → databricks.koalas.series.Series

Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

Analyzes both numeric and object series, as well as DataFrame column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

### Parameters

**percentiles** [list of float in range [0.0, 1.0], default [0.25, 0.5, 0.75]] A list of percentiles to be computed.

### Returns

**DataFrame** Summary statistics of the Dataframe provided.

See also:



**DataFrame.count** Count number of non-NA/null observations.

**DataFrame.max** Maximum of the values in the object.

**DataFrame.min** Minimum of the values in the object.

**DataFrame.mean** Mean of the values.

**DataFrame.std** Standard deviation of the observations.

## Notes

For numeric data, the result's index will include count, mean, std, min, 25%, 50%, 75%, max.

Currently only numeric data is supported.

## Examples

Describing a numeric Series.

```
>>> s = ks.Series([1, 2, 3])
>>> s.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.0
50%        2.0
75%        3.0
max        3.0
dtype: float64
```

Describing a DataFrame. Only numeric fields are returned.

```
>>> df = ks.DataFrame({'numeric1': [1, 2, 3],
...                    'numeric2': [4.0, 5.0, 6.0],
...                    'object': ['a', 'b', 'c']
...                    },
...                    columns=['numeric1', 'numeric2', 'object'])
>>> df.describe()
      numeric1  numeric2
count         3.0       3.0
mean          2.0       5.0
std           1.0       1.0
min           1.0       4.0
25%           1.0       4.0
50%           2.0       5.0
75%           3.0       6.0
max           3.0       6.0
```

For multi-index columns:

```
>>> df.columns = [('num', 'a'), ('num', 'b'), ('obj', 'c')]
>>> df.describe()
      num
      a  b
count 3.0 3.0
```

(continues on next page)

(continued from previous page)

```

mean    2.0    5.0
std     1.0    1.0
min     1.0    4.0
25%     1.0    4.0
50%     2.0    5.0
75%     3.0    6.0
max     3.0    6.0

```

```

>>> df[['num', 'b']].describe()
count    3.0
mean     5.0
std      1.0
min      4.0
25%      4.0
50%      5.0
75%      6.0
max      6.0
Name: (num, b), dtype: float64

```

Describing a DataFrame and selecting custom percentiles.

```

>>> df = ks.DataFrame({'numeric1': [1, 2, 3],
...                    'numeric2': [4.0, 5.0, 6.0]
...                    },
...                    columns=['numeric1', 'numeric2'])
>>> df.describe(percentiles = [0.85, 0.15])

```

	numeric1	numeric2
count	3.0	3.0
mean	2.0	5.0
std	1.0	1.0
min	1.0	4.0
15%	1.0	4.0
50%	2.0	5.0
85%	3.0	6.0
max	3.0	6.0

Describing a column from a DataFrame by accessing it as an attribute.

```

>>> df.numeric1.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.0
50%      2.0
75%      3.0
max      3.0
Name: numeric1, dtype: float64

```

Describing a column from a DataFrame by accessing it as an attribute and selecting custom percentiles.

```

>>> df.numeric1.describe(percentiles = [0.85, 0.15])
count    3.0
mean     2.0
std      1.0
min      1.0

```

(continues on next page)

(continued from previous page)

```

15%      1.0
50%      2.0
85%      3.0
max       3.0
Name: numeric1, dtype: float64

```

**databricks.koalas.Series.filter**

`Series.filter(items=None, like=None, regex=None, axis=None)` → `databricks.koalas.series.Series`

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

**Parameters**

**items** [list-like] Keep labels from axis which are in items.

**like** [string] Keep labels from axis for which “like in label == True”.

**regex** [string (regular expression)] Keep labels from axis for which `re.search(regex, label) == True`.

**axis** [int or string axis name] The axis to filter on. By default this is the info axis, ‘index’ for Series, ‘columns’ for DataFrame.

**Returns**

same type as input object

See also:

[`DataFrame.loc`](#)

**Notes**

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

**Examples**

```

>>> df = ks.DataFrame(np.array([[1, 2, 3], [4, 5, 6]]),
...                    index=['mouse', 'rabbit'],
...                    columns=['one', 'two', 'three'])

```

```

>>> # select columns by name
>>> df.filter(items=['one', 'three'])
      one  three
mouse    1     3
rabbit    4     6

```

```

>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
      one  three

```

(continues on next page)

(continued from previous page)

mouse	1	3
rabbit	4	6

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
      one  two  three
rabbit   4    5     6
```

For a Series,

```
>>> # select rows by name
>>> df.one.filter(items=['rabbit'])
rabbit    4
Name: one, dtype: int64
```

```
>>> # select rows by regular expression
>>> df.one.filter(regex='e$')
mouse     1
Name: one, dtype: int64
```

```
>>> # select rows containing 'bbi'
>>> df.one.filter(like='bbi')
rabbit    4
Name: one, dtype: int64
```

### databricks.koalas.Series.kurt

**Series.kurt** (*axis*: Union[int, str] = None, *numeric\_only*: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]  
 Return unbiased kurtosis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

#### Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

#### Returns

**kurt** [scalar for a Series, and a Series for a DataFrame.]

### Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                    columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.kurtosis()
a    -1.5
b    -1.5
dtype: float64
```

On a Series:

```
>>> df['a'].kurtosis()
-1.5
```

### databricks.koalas.Series.mad

Series.**mad**() → float

Return the mean absolute deviation of values.

#### Examples

```
>>> s = ks.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.mad()
1.0
```

### databricks.koalas.Series.max

Series.**max**(axis: Union[int, str] = None, numeric\_only: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]

Return the maximum of the values.

#### Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] If True, include only float, int, boolean columns. This parameter is mainly for pandas compatibility. False is supported; however, the columns should be all numeric or all non-numeric.

#### Returns

**max** [scalar for a Series, and a Series for a DataFrame.]

#### Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                    columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.max()
a    3.0
b    0.3
dtype: float64
```

```
>>> df.max(axis=1)
0    1.0
1    2.0
2    3.0
3    NaN
dtype: float64
```

On a Series:

```
>>> df['a'].max()
3.0
```

### **databricks.koalas.Series.mean**

`Series.mean(axis: Union[int, str] = None, numeric_only: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]`

Return the mean of the values.

#### **Parameters**

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

#### **Returns**

**mean** [scalar for a Series, and a Series for a DataFrame.]

### **Examples**

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                    columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.mean()
a    2.0
b    0.2
dtype: float64
```

```
>>> df.mean(axis=1)
0    0.55
1    1.10
2    1.65
3    NaN
dtype: float64
```

On a Series:

```
>>> df['a'].mean()
2.0
```

**databricks.koalas.Series.min**

`Series.min(axis: Union[int, str] = None, numeric_only: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]`  
 Return the minimum of the values.

**Parameters**

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] If True, include only float, int, boolean columns. This parameter is mainly for pandas compatibility. False is supported; however, the columns should be all numeric or all non-numeric.

**Returns**

**min** [scalar for a Series, and a Series for a DataFrame.]

**Examples**

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                    columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.min()
a    1.0
b    0.1
dtype: float64
```

```
>>> df.min(axis=1)
0    0.1
1    0.2
2    0.3
3    NaN
dtype: float64
```

On a Series:

```
>>> df['a'].min()
1.0
```

**databricks.koalas.Series.mode**

`Series.mode(dropna=True) → databricks.koalas.series.Series`  
 Return the mode(s) of the dataset.

Always returns Series even if only one value is returned.

**Parameters**

**dropna** [bool, default True] Don't consider counts of NaN/NaT.

**Returns**

**Series** Modes of the Series.

## Examples

```
>>> s = ks.Series([0, 0, 1, 1, 1, np.nan, np.nan, np.nan])
>>> s
0    0.0
1    0.0
2    1.0
3    1.0
4    1.0
5    NaN
6    NaN
7    NaN
dtype: float64
```

```
>>> s.mode()
0    1.0
dtype: float64
```

If there are several same modes, all items are shown

```
>>> s = ks.Series([0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3,
...                np.nan, np.nan, np.nan])
>>> s
0    0.0
1    0.0
2    1.0
3    1.0
4    1.0
5    2.0
6    2.0
7    2.0
8    3.0
9    3.0
10   3.0
11   NaN
12   NaN
13   NaN
dtype: float64
```

```
>>> s.mode().sort_values()
...  1.0
...  2.0
...  3.0
dtype: float64
```

With 'dropna' set to 'False', we can also see NaN in the result

```
>>> s.mode(dropna=False).sort_values()
...  1.0
...  2.0
...  3.0
...  NaN
dtype: float64
```



**databricks.koalas.Series.nlargest**

`Series.nlargest` (*n*: `int` = 5) → `databricks.koalas.series.Series`

Return the largest *n* elements.

**Parameters**

**n** [`int`, default 5]

**Returns**

**Series** The *n* largest values in the Series, sorted in decreasing order.

**See also:**

`Series.nsmallest` Get the *n* smallest elements.

`Series.sort_values` Sort Series by values.

`Series.head` Return the first *n* rows.

**Notes**

Faster than `.sort_values(ascending=False).head(n)` for small *n* relative to the size of the Series object.

In Koalas, thanks to Spark's lazy execution and query optimizer, the two would have same performance.

**Examples**

```
>>> data = [1, 2, 3, 4, np.nan, 6, 7, 8]
>>> s = ks.Series(data)
>>> s
0    1.0
1    2.0
2    3.0
3    4.0
4    NaN
5    6.0
6    7.0
7    8.0
dtype: float64
```

The *n* largest elements where *n*=5 by default.

```
>>> s.nlargest()
7    8.0
6    7.0
5    6.0
3    4.0
2    3.0
dtype: float64
```

```
>>> s.nlargest(n=3)
7    8.0
6    7.0
5    6.0
dtype: float64
```

**databricks.koalas.Series.nsmallest**

`Series.nsmallest` (*n*: *int* = 5) → `databricks.koalas.series.Series`

Return the smallest *n* elements.

**Parameters**

**n** [*int*, default 5] Return this many ascending sorted values.

**Returns**

**Series** The *n* smallest values in the Series, sorted in increasing order.

**See also:**

[`Series.nlargest`](#) Get the *n* largest elements.

[`Series.sort\_values`](#) Sort Series by values.

[`Series.head`](#) Return the first *n* rows.

**Notes**

Faster than `.sort_values().head(n)` for small *n* relative to the size of the `Series` object. In Koalas, thanks to Spark's lazy execution and query optimizer, the two would have same performance.

**Examples**

```
>>> data = [1, 2, 3, 4, np.nan, 6, 7, 8]
>>> s = ks.Series(data)
>>> s
0    1.0
1    2.0
2    3.0
3    4.0
4    NaN
5    6.0
6    7.0
7    8.0
dtype: float64
```

The *n* largest elements where *n*=5 by default.

```
>>> s.nsmallest()
0    1.0
1    2.0
2    3.0
3    4.0
5    6.0
dtype: float64
```

```
>>> s.nsmallest(3)
0    1.0
1    2.0
2    3.0
dtype: float64
```

**databricks.koalas.Series.pct\_change**

`Series.pct_change (periods=1)` → `databricks.koalas.series.Series`  
 Percentage change between the current and a prior element.

---

**Note:** the current implementation of this API uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

**Parameters**

**periods** [int, default 1] Periods to shift for forming percent change.

**Returns**

**Series**

**Examples**

```
>>> kser = ks.Series([90, 91, 85], index=[2, 4, 1])
>>> kser
2    90
4    91
1    85
dtype: int64
```

```
>>> kser.pct_change()
2      NaN
4    0.011111
1   -0.065934
dtype: float64
```

```
>>> kser.sort_index().pct_change()
1      NaN
2    0.058824
4    0.011111
dtype: float64
```

```
>>> kser.pct_change(periods=2)
2      NaN
4      NaN
1   -0.055556
dtype: float64
```

**databricks.koalas.Series.prod**

`Series.prod(axis: Union[int, str] = None, numeric_only: bool = None, min_count: int = 0) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]`  
 Return the product of the values.

**Note:** unlike pandas', Koalas' emulates product by `exp(sum(log(...)))` trick. Therefore, it only works for positive numbers.

**Parameters**

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**min\_count** [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

**Examples**

On a DataFrame:

Non-numeric type column is not included to the result.

```
>>> kdf = ks.DataFrame({'A': [1, 2, 3, 4, 5],
...                     'B': [10, 20, 30, 40, 50],
...                     'C': ['a', 'b', 'c', 'd', 'e']})
>>> kdf
   A  B  C
0  1 10  a
1  2 20  b
2  3 30  c
3  4 40  d
4  5 50  e
```

```
>>> kdf.prod()
A      120
B  12000000
dtype: int64
```

If there is no numeric type columns, returns empty Series.

```
>>> ks.DataFrame({"key": ['a', 'b', 'c'], "val": ['x', 'y', 'z']}).prod()
Series([], dtype: float64)
```

On a Series:

```
>>> ks.Series([1, 2, 3, 4, 5]).prod()
120
```

By default, the product of an empty or all-NA Series is 1

```
>>> ks.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> ks.Series([]).prod(min_count=1)
nan
```

### `databricks.koalas.Series.nunique`

`Series.nunique` (*dropna*: bool = True, *approx*: bool = False, *rsd*: float = 0.05) → int

Return number of unique elements in the object. Excludes NA values by default.

#### Parameters

**dropna** [bool, default True] Don't include NaN in the count.

**approx**: bool, default False If False, will use the exact algorithm and return the exact number of unique. If True, it uses the HyperLogLog approximate algorithm, which is significantly faster for large amount of data. Note: This parameter is specific to Koalas and is not found in pandas.

**rsd**: float, default 0.05 Maximum estimation error allowed in the HyperLogLog algorithm. Note: Just like `approx` this parameter is specific to Koalas.

#### Returns

int

See also:

`DataFrame.nunique` Method `nunique` for `DataFrame`.

`Series.count` Count non-NA/null observations in the Series.

### Examples

```
>>> ks.Series([1, 2, 3, np.nan]).nunique()
3
```

```
>>> ks.Series([1, 2, 3, np.nan]).nunique(dropna=False)
4
```

On big data, we recommend using the approximate algorithm to speed up this function. The result will be very close to the exact unique count.

```
>>> ks.Series([1, 2, 3, np.nan]).nunique(approx=True)
3
```

```
>>> idx = ks.Index([1, 1, 2, None])
>>> idx
Float64Index([1.0, 1.0, 2.0, nan], dtype='float64')
```

```
>>> idx.nunique()
2
```

```
>>> idx.nunique(dropna=False)
3
```

**databricks.koalas.Series.is\_unique****property** `Series.is_unique`

Return boolean if values in the object are unique

**Returns**

**is\_unique** [boolean]

```
>>> ks.Series([1, 2, 3]).is_unique
..
```

**True**

```
>>> ks.Series([1, 2, 2]).is_unique
..
```

**False**

```
>>> ks.Series([1, 2, 3, None]).is_unique
..
```

**True****databricks.koalas.Series.quantile**

`Series.quantile` (*q*: `Union[float, Iterable[float]] = 0.5`, *accuracy*: `int = 10000`) → `Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]`  
Return value at the given quantile.

---

**Note:** Unlike pandas', the quantile in Koalas is an approximated quantile based upon approximate percentile computation because computing quantile across a large dataset is extremely expensive.

---

**Parameters**

**q** [float or array-like, default 0.5 (50% quantile)] 0 <= q <= 1, the quantile(s) to compute.

**accuracy** [int, optional] Default accuracy of approximation. Larger value means better accuracy. The relative error can be deduced by 1.0 / accuracy.

**Returns**

**float or Series** If the current object is a Series and *q* is an array, a Series will be returned where the index is *q* and the values are the quantiles, otherwise a float will be returned.

## Examples

```
>>> s = ks.Series([1, 2, 3, 4, 5])
>>> s.quantile(.5)
3.0
```

```
>>> (s + 1).quantile(.5)
4.0
```

```
>>> s.quantile([.25, .5, .75])
0.25    2.0
0.50    3.0
0.75    4.0
dtype: float64
```

```
>>> (s + 1).quantile([.25, .5, .75])
0.25    3.0
0.50    4.0
0.75    5.0
dtype: float64
```

## databricks.koalas.Series.rank

`Series.rank` (*method='average', ascending=True*) → `databricks.koalas.series.Series`

Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values.

---

**Note:** the current implementation of rank uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

### Parameters

**method** [{ 'average', 'min', 'max', 'first', 'dense' }]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

**ascending** [boolean, default True] False for ranks by high (1) to low (N)

### Returns

**ranks** [same type as caller]

## Examples

```
>>> s = ks.Series([1, 2, 2, 3], name='A')
>>> s
0    1
1    2
2    2
3    3
Name: A, dtype: int64
```

```
>>> s.rank()
0    1.0
1    2.5
2    2.5
3    4.0
Name: A, dtype: float64
```

If method is set to 'min', it use lowest rank in group.

```
>>> s.rank(method='min')
0    1.0
1    2.0
2    2.0
3    4.0
Name: A, dtype: float64
```

If method is set to 'max', it use highest rank in group.

```
>>> s.rank(method='max')
0    1.0
1    3.0
2    3.0
3    4.0
Name: A, dtype: float64
```

If method is set to 'first', it is assigned rank in order without groups.

```
>>> s.rank(method='first')
0    1.0
1    2.0
2    3.0
3    4.0
Name: A, dtype: float64
```

If method is set to 'dense', it leaves no gaps in group.

```
>>> s.rank(method='dense')
0    1.0
1    2.0
2    2.0
3    3.0
Name: A, dtype: float64
```



**databricks.koalas.Series.sem**

`Series.sem(axis: Union[int, str] = None, ddof: int = 1, numeric_only: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]`

Return unbiased standard error of the mean over requested axis.

**Parameters**

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**Returns**

**scalar(for Series) or Series(for DataFrame)**

**Examples**

```
>>> kdf = ks.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
>>> kdf
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> kdf.sem()
a    0.57735
b    0.57735
dtype: float64
```

```
>>> kdf.sem(ddof=0)
a    0.471405
b    0.471405
dtype: float64
```

```
>>> kdf.sem(axis=1)
0    1.5
1    1.5
2    1.5
dtype: float64
```

**Support for Series**

```
>>> kser = kdf.a
>>> kser
0    1
1    2
2    3
Name: a, dtype: int64
```

```
>>> kser.sem()
0.5773502691896258
```

```
>>> kser.sem(ddof=0)
0.47140452079103173
```

### **databricks.koalas.Series.skew**

**Series.skew** (*axis*: Union[int, str] = None, *numeric\_only*: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]  
Return unbiased skew normalized by N-1.

#### **Parameters**

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

#### **Returns**

**skew** [scalar for a Series, and a Series for a DataFrame.]

### **Examples**

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                    columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.skew()
a    0.000000e+00
b   -3.319678e-16
dtype: float64
```

On a Series:

```
>>> df['a'].skew()
0.0
```

### **databricks.koalas.Series.std**

**Series.std** (*axis*: Union[int, str] = None, *ddof*: int = 1, *numeric\_only*: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]  
Return sample standard deviation.

#### **Parameters**

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

#### **Returns**

**std** [scalar for a Series, and a Series for a DataFrame.]

## Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                     columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.std()
a    1.0
b    0.1
dtype: float64
```

```
>>> df.std(axis=1)
0    0.636396
1    1.272792
2    1.909188
3         NaN
dtype: float64
```

```
>>> df.std(ddof=0)
a    0.816497
b    0.081650
dtype: float64
```

On a Series:

```
>>> df['a'].std()
1.0
```

```
>>> df['a'].std(ddof=0)
0.816496580927726
```

## databricks.koalas.Series.sum

`Series.sum(axis: Union[int, str] = None, numeric_only: bool = None, min_count: int = 0) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]`  
Return the sum of the values.

### Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**min\_count** [int, default 0]

**The required number of valid values to perform the operation. If fewer than min\_count non-NA values are present the result will be NA.**

### Returns

**sum** [scalar for a Series, and a Series for a DataFrame.]

## Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, np.nan, 0.3, np.nan]},
...                     columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.sum()
a      6.0
b      0.4
dtype: float64
```

```
>>> df.sum(axis=1)
0      1.1
1      2.0
2      3.3
3      0.0
dtype: float64
```

```
>>> df.sum(min_count=3)
a      6.0
b      NaN
dtype: float64
```

```
>>> df.sum(axis=1, min_count=1)
0      1.1
1      2.0
2      3.3
3      NaN
dtype: float64
```

On a Series:

```
>>> df['a'].sum()
6.0
```

```
>>> df['a'].sum(min_count=3)
6.0
>>> df['b'].sum(min_count=3)
nan
```

## databricks.koalas.Series.median

`Series.median`(axis: *Union[int, str] = None*, *numeric\_only: bool = None*, *accuracy: int = 10000*) → *Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]*

Return the median of the values for the requested axis.

---

**Note:** Unlike pandas', the median in Koalas is an approximated median based upon approximate percentile computation because computing median across a large dataset is extremely expensive.

---

## Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**accuracy** [int, optional] Default accuracy of approximation. Larger value means better accuracy. The relative error can be deduced by  $1.0 / \text{accuracy}$ .

### Returns

**median** [scalar or Series]

### Examples

```
>>> df = ks.DataFrame({
...     'a': [24., 21., 25., 33., 26.], 'b': [1, 2, 3, 4, 5]}, columns=['a', 'b'])
>>> df
   a  b
0 24.0 1
1 21.0 2
2 25.0 3
3 33.0 4
4 26.0 5
```

On a DataFrame:

```
>>> df.median()
a    25.0
b     3.0
dtype: float64
```

On a Series:

```
>>> df['a'].median()
25.0
>>> (df['b'] + 100).median()
103.0
```

For multi-index columns,

```
>>> df.columns = pd.MultiIndex.from_tuples([('x', 'a'), ('y', 'b')])
>>> df
   x  y
   a  b
0 24.0 1
1 21.0 2
2 25.0 3
3 33.0 4
4 26.0 5
```

On a DataFrame:

```
>>> df.median()
x a    25.0
y b     3.0
dtype: float64
```

```
>>> df.median(axis=1)
0    12.5
1    11.5
2    14.0
3    18.5
4    15.5
dtype: float64
```

On a Series:

```
>>> df[('x', 'a')].median()
25.0
>>> (df[('y', 'b')] + 100).median()
103.0
```

### databricks.koalas.Series.var

`Series.var` (*axis*: Union[int, str] = None, *ddof*: int = 1, *numeric\_only*: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]  
Return unbiased variance.

#### Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

#### Returns

**var** [scalar for a Series, and a Series for a DataFrame.]

### Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                    columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.var()
a    1.00
b    0.01
dtype: float64
```

```
>>> df.var(axis=1)
0    0.405
1    1.620
2    3.645
3     NaN
dtype: float64
```

```
>>> df.var(ddof=0)
a    0.666667
b    0.006667
dtype: float64
```

On a Series:

```
>>> df['a'].var()
1.0
```

```
>>> df['a'].var(ddof=0)
0.6666666666666666
```

### databricks.koalas.Series.kurtosis

`Series.kurtosis` (*axis*: Union[int, str] = None, *numeric\_only*: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]

Return unbiased kurtosis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

#### Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

#### Returns

**kurt** [scalar for a Series, and a Series for a DataFrame.]

### Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                    columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.kurtosis()
a    -1.5
b    -1.5
dtype: float64
```

On a Series:

```
>>> df['a'].kurtosis()
-1.5
```

## `databricks.koalas.Series.unique`

`Series.unique()` → `databricks.koalas.series.Series`

Return unique values of Series object.

Uniques are returned in order of appearance. Hash table-based unique, therefore does NOT sort.

---

**Note:** This method returns newly created Series whereas pandas returns the unique values as a NumPy array.

---

### Returns

Returns the unique values as a Series.

See also:

*`Index.unique`*

*`groupby.SeriesGroupBy.unique`*

### Examples

```
>>> kser = ks.Series([2, 1, 3, 3], name='A')
>>> kser.unique().sort_values()

... 1
... 2
... 3
Name: A, dtype: int64
```

```
>>> ks.Series([pd.Timestamp('2016-01-01') for _ in range(3)]).unique()
0    2016-01-01
dtype: datetime64[ns]
```

```
>>> kser.name = ('x', 'a')
>>> kser.unique().sort_values()

... 1
... 2
... 3
Name: (x, a), dtype: int64
```

## `databricks.koalas.Series.value_counts`

`Series.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)` → `Series`

Return a Series containing counts of unique values. The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

### Parameters

**normalize** [boolean, default False] If True then the object returned will contain the relative frequencies of the unique values.

**sort** [boolean, default True] Sort by values.



**ascending** [boolean, default False] Sort in ascending order.

**bins** [Not Yet Supported]

**dropna** [boolean, default True] Don't include counts of NaN.

### Returns

**counts** [Series]

### See also:

[\*Series.count\*](#) Number of non-NA elements in a Series.

## Examples

### For Series

```
>>> df = ks.DataFrame({'x': [0, 0, 1, 1, 1, np.nan]})
>>> df.x.value_counts()
1.0    3
0.0    2
Name: x, dtype: int64
```

With *normalize* set to *True*, returns the relative frequency by dividing all values by the sum of values.

```
>>> df.x.value_counts(normalize=True)
1.0    0.6
0.0    0.4
Name: x, dtype: float64
```

**dropna** With *dropna* set to *False* we can also see NaN index values.

```
>>> df.x.value_counts(dropna=False)
1.0    3
0.0    2
NaN    1
Name: x, dtype: int64
```

### For Index

```
>>> idx = ks.Index([3, 1, 2, 3, 4, np.nan])
>>> idx
Float64Index([3.0, 1.0, 2.0, 3.0, 4.0, nan], dtype='float64')
```

```
>>> idx.value_counts().sort_index()
1.0    1
2.0    1
3.0    2
4.0    1
dtype: int64
```

### sort

With *sort* set to *False*, the result wouldn't be sorted by number of count.

```
>>> idx.value_counts(sort=True).sort_index()
1.0    1
2.0    1
3.0    2
4.0    1
dtype: int64
```

**normalize**

With *normalize* set to *True*, returns the relative frequency by dividing all values by the sum of values.

```
>>> idx.value_counts(normalize=True).sort_index()
1.0    0.2
2.0    0.2
3.0    0.4
4.0    0.2
dtype: float64
```

**dropna**

With *dropna* set to *False* we can also see NaN index values.

```
>>> idx.value_counts(dropna=False).sort_index()
1.0    1
2.0    1
3.0    2
4.0    1
NaN    1
dtype: int64
```

For MultiIndex.

```
>>> midx = pd.MultiIndex([['lama', 'cow', 'falcon'],
...                       ['speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [1, 1, 1, 1, 1, 2, 1, 2, 2]])
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3], index=midx)
>>> s.index
MultiIndex([( 'lama', 'weight'),
            ( 'lama', 'weight'),
            ( 'lama', 'weight'),
            ( 'cow', 'weight'),
            ( 'cow', 'weight'),
            ( 'cow', 'length'),
            ('falcon', 'weight'),
            ('falcon', 'length'),
            ('falcon', 'length')],
            )
```

```
>>> s.index.value_counts().sort_index()
(cow, length)    1
(cow, weight)    2
(falcon, length)  2
(falcon, weight)  1
(lama, weight)    3
dtype: int64
```

```
>>> s.index.value_counts(normalize=True).sort_index()
(cow, length)      0.111111
(cow, weight)      0.222222
(falcon, length)   0.222222
(falcon, weight)   0.111111
(lama, weight)     0.333333
dtype: float64
```

If Index has name, keep the name up.

```
>>> idx = ks.Index([0, 0, 0, 1, 1, 2, 3], name='koalas')
>>> idx.value_counts().sort_index()
0      3
1      2
2      1
3      1
Name: koalas, dtype: int64
```

### **databricks.koalas.Series.round**

`Series.round(decimals=0) → databricks.koalas.series.Series`

Round each value in a Series to the given number of decimals.

#### **Parameters**

**decimals** [int] Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

#### **Returns**

**Series object**

**See also:**

[\*DataFrame.round\*](#)

### **Examples**

```
>>> df = ks.Series([0.028208, 0.038683, 0.877076], name='x')
>>> df
0      0.028208
1      0.038683
2      0.877076
Name: x, dtype: float64
```

```
>>> df.round(2)
0      0.03
1      0.04
2      0.88
Name: x, dtype: float64
```

**databricks.koalas.Series.diff**

`Series.diff( periods=1 )` → `databricks.koalas.series.Series`

First discrete difference of element.

Calculates the difference of a Series element compared with another element in the DataFrame (default is the element in the same column of the previous row).

---

**Note:** the current implementation of `diff` uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

**Parameters**

**periods** [int, default 1] Periods to shift for calculating difference, accepts negative values.

**Returns**

**diffed** [Series]

**Examples**

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]}), columns=['a', 'b', 'c'])
>>> df
   a  b  c
0  1  1  1
1  2  1  4
2  3  2  9
3  4  3 16
4  5  5 25
5  6  8 36
```

```
>>> df.b.diff()
0    NaN
1    0.0
2    1.0
3    1.0
4    2.0
5    3.0
Name: b, dtype: float64
```

Difference with previous value

```
>>> df.c.diff(periods=3)
0    NaN
1    NaN
2    NaN
3   15.0
4   21.0
5   27.0
Name: c, dtype: float64
```

Difference with following value

```
>>> df.c.diff(periods=-1)
0      -3.0
1      -5.0
2      -7.0
3      -9.0
4     -11.0
5         NaN
Name: c, dtype: float64
```

### `databricks.koalas.Series.is_monotonic`

#### **property** `Series.is_monotonic`

Return boolean if values in the object are monotonically increasing.

**Note:** the current implementation of `is_monotonic` requires to shuffle and aggregate multiple times to check the order locally and globally, which is potentially expensive. In case of multi-index, all data are transferred to single node which can easily cause out-of-memory error currently.

**Note:** Disable the Spark config `spark.sql.optimizer.nestedSchemaPruning.enabled` for multi-index if you're using Koalas < 1.7.0 with PySpark 3.1.1.

#### Returns

`is_monotonic` [bool]

#### Examples

```
>>> ser = ks.Series(['1/1/2018', '3/1/2018', '4/1/2018'])
>>> ser.is_monotonic
True
```

```
>>> df = ks.DataFrame({'dates': [None, '1/1/2018', '2/1/2018', '3/1/2018']})
>>> df.dates.is_monotonic
False
```

```
>>> df.index.is_monotonic
True
```

```
>>> ser = ks.Series([1])
>>> ser.is_monotonic
True
```

```
>>> ser = ks.Series([])
>>> ser.is_monotonic
True
```

```
>>> ser.rename("a").to_frame().set_index("a").index.is_monotonic
True
```

```
>>> ser = ks.Series([5, 4, 3, 2, 1], index=[1, 2, 3, 4, 5])
>>> ser.is_monotonic
False
```

```
>>> ser.index.is_monotonic
True
```

### Support for MultiIndex

```
>>> midx = ks.MultiIndex.from_tuples(
... [ ('x', 'a'), ('x', 'b'), ('y', 'c'), ('y', 'd'), ('z', 'e') ])
>>> midx
MultiIndex([ ('x', 'a'),
              ('x', 'b'),
              ('y', 'c'),
              ('y', 'd'),
              ('z', 'e') ],
           )
>>> midx.is_monotonic
True
```

```
>>> midx = ks.MultiIndex.from_tuples(
... [ ('z', 'a'), ('z', 'b'), ('y', 'c'), ('y', 'd'), ('x', 'e') ])
>>> midx
MultiIndex([ ('z', 'a'),
              ('z', 'b'),
              ('y', 'c'),
              ('y', 'd'),
              ('x', 'e') ],
           )
>>> midx.is_monotonic
False
```

### `databricks.koalas.Series.is_monotonic_increasing`

#### **property** `Series.is_monotonic_increasing`

Return boolean if values in the object are monotonically increasing.

---

**Note:** the current implementation of `is_monotonic` requires to shuffle and aggregate multiple times to check the order locally and globally, which is potentially expensive. In case of multi-index, all data are transferred to single node which can easily cause out-of-memory error currently.

---

---

**Note:** Disable the Spark config `spark.sql.optimizer.nestedSchemaPruning.enabled` for multi-index if you're using Koalas < 1.7.0 with PySpark 3.1.1.

---

#### **Returns**

`is_monotonic` [bool]

## Examples

```
>>> ser = ks.Series(['1/1/2018', '3/1/2018', '4/1/2018'])
>>> ser.is_monotonic
True
```

```
>>> df = ks.DataFrame({'dates': [None, '1/1/2018', '2/1/2018', '3/1/2018']})
>>> df.dates.is_monotonic
False
```

```
>>> df.index.is_monotonic
True
```

```
>>> ser = ks.Series([1])
>>> ser.is_monotonic
True
```

```
>>> ser = ks.Series([])
>>> ser.is_monotonic
True
```

```
>>> ser.rename("a").to_frame().set_index("a").index.is_monotonic
True
```

```
>>> ser = ks.Series([5, 4, 3, 2, 1], index=[1, 2, 3, 4, 5])
>>> ser.is_monotonic
False
```

```
>>> ser.index.is_monotonic
True
```

## Support for MultiIndex

```
>>> midx = ks.MultiIndex.from_tuples(
...   [('x', 'a'), ('x', 'b'), ('y', 'c'), ('y', 'd'), ('z', 'e')])
>>> midx
MultiIndex([('x', 'a'),
            ('x', 'b'),
            ('y', 'c'),
            ('y', 'd'),
            ('z', 'e')],
           )
>>> midx.is_monotonic
True
```

```
>>> midx = ks.MultiIndex.from_tuples(
...   [('z', 'a'), ('z', 'b'), ('y', 'c'), ('y', 'd'), ('x', 'e')])
>>> midx
MultiIndex([('z', 'a'),
            ('z', 'b'),
            ('y', 'c'),
            ('y', 'd'),
            ('x', 'e')],
           )
```

(continues on next page)

(continued from previous page)

```
>>> midx.is_monotonic
False
```

## databricks.koalas.Series.is\_monotonic\_decreasing

### property Series.is\_monotonic\_decreasing

Return boolean if values in the object are monotonically decreasing.

**Note:** the current implementation of `is_monotonic_decreasing` requires to shuffle and aggregate multiple times to check the order locally and globally, which is potentially expensive. In case of multi-index, all data are transferred to single node which can easily cause out-of-memory error currently.

**Note:** Disable the Spark config `spark.sql.optimizer.nestedSchemaPruning.enabled` for multi-index if you're using Koalas < 1.7.0 with PySpark 3.1.1.

### Returns

`is_monotonic` [bool]

### Examples

```
>>> ser = ks.Series(['4/1/2018', '3/1/2018', '1/1/2018'])
>>> ser.is_monotonic_decreasing
True
```

```
>>> df = ks.DataFrame({'dates': [None, '3/1/2018', '2/1/2018', '1/1/2018']})
>>> df.dates.is_monotonic_decreasing
False
```

```
>>> df.index.is_monotonic_decreasing
False
```

```
>>> ser = ks.Series([1])
>>> ser.is_monotonic_decreasing
True
```

```
>>> ser = ks.Series([])
>>> ser.is_monotonic_decreasing
True
```

```
>>> ser.rename("a").to_frame().set_index("a").index.is_monotonic_decreasing
True
```

```
>>> ser = ks.Series([5, 4, 3, 2, 1], index=[1, 2, 3, 4, 5])
>>> ser.is_monotonic_decreasing
True
```



```
>>> ser.index.is_monotonic_decreasing
False
```

### Support for MultiIndex

```
>>> midx = ks.MultiIndex.from_tuples(
...     [('x', 'a'), ('x', 'b'), ('y', 'c'), ('y', 'd'), ('z', 'e')])
>>> midx
MultiIndex([('x', 'a'),
            ('x', 'b'),
            ('y', 'c'),
            ('y', 'd'),
            ('z', 'e')],
           )
>>> midx.is_monotonic_decreasing
False
```

```
>>> midx = ks.MultiIndex.from_tuples(
...     [('z', 'e'), ('z', 'd'), ('y', 'c'), ('y', 'b'), ('x', 'a')])
>>> midx
MultiIndex([('z', 'a'),
            ('z', 'b'),
            ('y', 'c'),
            ('y', 'd'),
            ('x', 'e')],
           )
>>> midx.is_monotonic_decreasing
True
```

## 3.3.8 Reindexing / Selection / Label manipulation

<code>Series.align(other[, join, axis, copy])</code>	Align two objects on their axes with the specified join method.
<code>Series.drop([labels, index, level])</code>	Return Series with specified index labels removed.
<code>Series.droplevel(level)</code>	Return Series with requested index level(s) removed.
<code>Series.drop_duplicates([keep, inplace])</code>	Return Series with duplicate values removed.
<code>Series.equals(other)</code>	Compare if the current value is equal to the other.
<code>Series.add_prefix(prefix)</code>	Prefix labels with string <i>prefix</i> .
<code>Series.add_suffix(suffix)</code>	Suffix labels with string <i>suffix</i> .
<code>Series.first(offset)</code>	Select first periods of time series data based on a date offset.
<code>Series.head([n])</code>	Return the first n rows.
<code>Series.idxmax([skipna])</code>	Return the row label of the maximum value.
<code>Series.idxmin([skipna])</code>	Return the row label of the minimum value.
<code>Series.isin(values)</code>	Check whether <i>values</i> are contained in Series or Index.
<code>Series.last(offset)</code>	Select final periods of time series data based on a date offset.
<code>Series.rename([index])</code>	Alter Series name.
<code>Series.rename_axis([mapper, index, inplace])</code>	Set the name of the axis for the index or columns.
<code>Series.reindex([index, fill_value])</code>	Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.

continues on next page

Table 26 – continued from previous page

<code>Series.reindex_like(other)</code>	Return a Series with matching indices as other object.
<code>Series.reset_index([level, drop, name, inplace])</code>	Generate a new DataFrame or Series with the index reset.
<code>Series.sample([n, frac, replace, random_state])</code>	Return a random sample of items from an axis of object.
<code>Series.swaplevel([i, j, copy])</code>	Swap levels i and j in a MultiIndex.
<code>Series.swapaxes(i, j[, copy])</code>	Interchange axes and swap values axes appropriately.
<code>Series.take(indices)</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>Series.tail([n])</code>	Return the last <i>n</i> rows.
<code>Series.where(cond[, other])</code>	Replace values where the condition is False.
<code>Series.mask(cond[, other])</code>	Replace values where the condition is True.
<code>Series.truncate([before, after, axis, copy])</code>	Truncate a Series or DataFrame before and after some index value.

**databricks.koalas.Series.align**

`Series.align` (*other*: Union[databricks.koalas.frame.DataFrame, Series], *join*: str = 'outer', *axis*: Union[str, int, None] = None, *copy*: bool = True) → Tuple[databricks.koalas.series.Series, Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]]

Align two objects on their axes with the specified join method.

Join method is specified for each axis Index.

**Parameters**

**other** [DataFrame or Series]

**join** [{ 'outer', 'inner', 'left', 'right' }, default 'outer']

**axis** [allowed axis of the other object, default None] Align on index (0), columns (1), or both (None).

**copy** [bool, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

**Returns**

(**left**, **right**) [(Series, type of other)] Aligned objects.

**Examples**

```
>>> ks.set_option("compute.ops_on_diff_frames", True)
>>> s1 = ks.Series([7, 8, 9], index=[10, 11, 12])
>>> s2 = ks.Series(["g", "h", "i"], index=[10, 20, 30])
```

```
>>> aligned_l, aligned_r = s1.align(s2)
>>> aligned_l.sort_index()
10    7.0
11    8.0
12    9.0
20    NaN
30    NaN
dtype: float64
>>> aligned_r.sort_index()
10    g
```

(continues on next page)

(continued from previous page)

```

11    None
12    None
20     h
30     i
dtype: object

```

Align with the join type “inner”:

```

>>> aligned_l, aligned_r = s1.align(s2, join="inner")
>>> aligned_l.sort_index()
10     7
dtype: int64
>>> aligned_r.sort_index()
10     g
dtype: object

```

Align with a DataFrame:

```

>>> df = ks.DataFrame({"a": [1, 2, 3], "b": ["a", "b", "c"]}, index=[10, 20, 30])
>>> aligned_l, aligned_r = s1.align(df)
>>> aligned_l.sort_index()
10     7.0
11     8.0
12     9.0
20    NaN
30    NaN
dtype: float64
>>> aligned_r.sort_index()
      a      b
10  1.0    a
11  NaN  None
12  NaN  None
20  2.0    b
30  3.0    c

```

```

>>> ks.reset_option("compute.ops_on_diff_frames")

```

### **databricks.koalas.Series.drop**

`Series.drop(labels=None, index: Union[Any, Tuple, List[Any], List[Tuple]] = None, level=None) → databricks.koalas.series.Series`  
 Return Series with specified index labels removed.

Remove elements of a Series based on specifying the index labels. When using a multi-index, labels on different levels can be removed by specifying the level.

#### **Parameters**

**labels** [single label or list-like] Index labels to drop.

**index** [None] Redundant for application on Series, but index can be used instead of labels.

**level** [int or level name, optional] For MultiIndex, level for which the labels will be removed.

#### **Returns**

**Series** Series with specified index labels removed.

See also:

*[Series.dropna](#)*

## Examples

```
>>> s = ks.Series(data=np.arange(3), index=['A', 'B', 'C'])
>>> s
A    0
B    1
C    2
dtype: int64
```

Drop single label A

```
>>> s.drop('A')
B    1
C    2
dtype: int64
```

Drop labels B and C

```
>>> s.drop(labels=['B', 'C'])
A    0
dtype: int64
```

With 'index' rather than 'labels' returns exactly same result.

```
>>> s.drop(index='A')
B    1
C    2
dtype: int64
```

```
>>> s.drop(index=['B', 'C'])
A    0
dtype: int64
```

Also support for MultiIndex

```
>>> midx = pd.MultiIndex([[ 'lama', 'cow', 'falcon'],
...                       [ 'speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3],
...               index=midx)
>>> s
lama    speed    45.0
        weight   200.0
        length    1.2
cow     speed    30.0
        weight   250.0
        length    1.5
falcon  speed    320.0
        weight    1.0
        length    0.3
dtype: float64
```

```
>>> s.drop(labels='weight', level=1)
lama      speed      45.0
          length      1.2
cow       speed      30.0
          length      1.5
falcon    speed     320.0
          length      0.3
dtype: float64
```

```
>>> s.drop(['lama', 'weight'])
lama      speed      45.0
          length      1.2
cow       speed      30.0
          weight     250.0
          length      1.5
falcon    speed     320.0
          weight      1.0
          length      0.3
dtype: float64
```

```
>>> s.drop([('lama', 'speed'), ('falcon', 'weight')])
lama      weight     200.0
          length      1.2
cow       speed      30.0
          weight     250.0
          length      1.5
falcon    speed     320.0
          length      0.3
dtype: float64
```

### databricks.koalas.Series.droplevel

`Series.droplevel (level)` → `databricks.koalas.series.Series`

Return Series with requested index level(s) removed.

#### Parameters

**level** [int, str, or list-like] If a string is given, must be the name of a level If list-like, elements must be names or positional indexes of levels.

#### Returns

**Series** Series with requested index level(s) removed.

### Examples

```
>>> kser = ks.Series(
...     [1, 2, 3],
...     index=pd.MultiIndex.from_tuples(
...         [("x", "a"), ("x", "b"), ("y", "c")], names=["level_1", "level_2"]
...     ),
... )
>>> kser
level_1 level_2
x      a      1
```

(continues on next page)

(continued from previous page)

```

      b      2
y      c      3
dtype: int64

```

#### Removing specific index level by level

```

>>> kser.droplevel(0)
level_2
a      1
b      2
c      3
dtype: int64

```

#### Removing specific index level by name

```

>>> kser.droplevel("level_2")
level_1
x      1
x      2
y      3
dtype: int64

```

### **databricks.koalas.Series.drop\_duplicates**

**Series.drop\_duplicates** (*keep='first', inplace=False*) → Optional[databricks.koalas.series.Series]

Return Series with duplicate values removed.

#### Parameters

**keep** [{‘first’, ‘last’, False}, default ‘first’] Method to handle dropping duplicates: - ‘first’ : Drop duplicates except for the first occurrence. - ‘last’ : Drop duplicates except for the last occurrence. - False : Drop all duplicates.

**inplace** [bool, default False] If True, performs operation inplace and returns None.

#### Returns

**Series** Series with duplicates dropped.

### Examples

Generate a Series with duplicated entries.

```

>>> s = ks.Series(['lama', 'cow', 'lama', 'beetle', 'lama', 'hippo'],
...               name='animal')
>>> s.sort_index()
0      lama
1      cow
2      lama
3  beetle
4      lama
5      hippo
Name: animal, dtype: object

```

With the ‘keep’ parameter, the selection behaviour of duplicated values can be changed. The value ‘first’ keeps the first occurrence for each set of duplicated entries. The default value of keep is ‘first’.

```
>>> s.drop_duplicates().sort_index()
0      lama
1      cow
3    beetle
5     hippo
Name: animal, dtype: object
```

The value 'last' for parameter 'keep' keeps the last occurrence for each set of duplicated entries.

```
>>> s.drop_duplicates(keep='last').sort_index()
1      cow
3    beetle
4      lama
5     hippo
Name: animal, dtype: object
```

The value False for parameter 'keep' discards all sets of duplicated entries. Setting the value of 'inplace' to True performs the operation inplace and returns None.

```
>>> s.drop_duplicates(keep=False, inplace=True)
>>> s.sort_index()
1      cow
3    beetle
5     hippo
Name: animal, dtype: object
```

### **databricks.koalas.Series.equals**

`Series.equals(other) → bool`

Compare if the current value is equal to the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                    'b': [1, np.nan, 1, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.a == 1
a      True
b     False
c     False
d     False
Name: a, dtype: bool
```

```
>>> df.b.eq(1)
a      True
b     False
c      True
d     False
Name: b, dtype: bool
```

**databricks.koalas.Series.add\_prefix**

`Series.add_prefix(prefix)` → `databricks.koalas.series.Series`  
Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

**Parameters**

**prefix** [str] The string to add before each label.

**Returns**

**Series** New Series with updated labels.

**See also:**

**`Series.add_suffix`** Suffix column labels with string *suffix*.

**`DataFrame.add_suffix`** Suffix column labels with string *suffix*.

**`DataFrame.add_prefix`** Prefix column labels with string *prefix*.

**Examples**

```
>>> s = ks.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

**databricks.koalas.Series.add\_suffix**

`Series.add_suffix(suffix)` → `databricks.koalas.series.Series`  
Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

**Parameters**

**suffix** [str] The string to add after each label.

**Returns**

**Series** New Series with updated labels.

**See also:**

**`Series.add_prefix`** Prefix row labels with string *prefix*.

**`DataFrame.add_prefix`** Prefix column labels with string *prefix*.



**DataFrame.add\_suffix** Suffix column labels with string *suffix*.

### Examples

```
>>> s = ks.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

### databricks.koalas.Series.first

**Series.first** (*offset: Union[str, pandas.\_libs.tslibs.offsets.DateOffset]*) → databricks.koalas.series.Series  
Select first periods of time series data based on a date offset.

When having a Series with dates as index, this function can select the first few elements based on a date offset.

#### Parameters

**offset** [str or DateOffset] The offset length of the data that will be selected. For instance, '3D' will display all the rows having their index within the first 3 days.

#### Returns

**Series** A subset of the caller.

#### Raises

**TypeError** If the index is not a *DatetimeIndex*

### Examples

```
>>> index = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> kser = ks.Series([1, 2, 3, 4], index=index)
>>> kser
2018-04-09    1
2018-04-11    2
2018-04-13    3
2018-04-15    4
dtype: int64
```

Get the rows for the first 3 days:

```
>>> kser.first('3D')
2018-04-09    1
2018-04-11    2
dtype: int64
```

Notice the data for 3 first calendar days were returned, not the first 3 observed days in the dataset, and therefore data for 2018-04-13 was not returned.

### **databricks.koalas.Series.head**

`Series.head(n: int = 5) → databricks.koalas.series.Series`

Return the first n rows.

This function returns the first n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

#### **Parameters**

**n** [Integer, default = 5]

#### **Returns**

**The first n rows of the caller object.**

### **Examples**

```
>>> df = ks.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion']})
>>> df.animal.head(2)
0    alligator
1         bee
Name: animal, dtype: object
```

### **databricks.koalas.Series.idxmax**

`Series.idxmax(skipna=True) → Union[Tuple, Any]`

Return the row label of the maximum value.

If multiple values equal the maximum, the first row label with that value is returned.

#### **Parameters**

**skipna** [bool, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

#### **Returns**

**Index** Label of the maximum value.

#### **Raises**

**ValueError** If the Series is empty.

**See also:**

[`Series.idxmin`](#) Return index *label* of the first occurrence of minimum of values.

## Examples

```
>>> s = ks.Series(data=[1, None, 4, 3, 5],
...               index=['A', 'B', 'C', 'D', 'E'])
>>> s
A    1.0
B    NaN
C    4.0
D    3.0
E    5.0
dtype: float64
```

```
>>> s.idxmax()
'E'
```

If *skipna* is False and there is an NA value in the data, the function returns nan.

```
>>> s.idxmax(skipna=False)
nan
```

In case of multi-index, you get a tuple:

```
>>> index = pd.MultiIndex.from_arrays([
...     ['a', 'a', 'b', 'b'], ['c', 'd', 'e', 'f']], names=('first', 'second'))
>>> s = ks.Series(data=[1, None, 4, 5], index=index)
>>> s
first second
a      c      1.0
      d      NaN
b      e      4.0
      f      5.0
dtype: float64
```

```
>>> s.idxmax()
('b', 'f')
```

If multiple values equal the maximum, the first row label with that value is returned.

```
>>> s = ks.Series([1, 100, 1, 100, 1, 100], index=[10, 3, 5, 2, 1, 8])
>>> s
10    1
3    100
5    1
2    100
1    1
8    100
dtype: int64
```

```
>>> s.idxmax()
3
```

**databricks.koalas.Series.idxmin**

`Series.idxmin(skipna=True) → Union[Tuple, Any]`

Return the row label of the minimum value.

If multiple values equal the minimum, the first row label with that value is returned.

**Parameters**

**skipna** [bool, default True] Exclude NA/null values. If the entire Series is NA, the result will be NA.

**Returns**

**Index** Label of the minimum value.

**Raises**

**ValueError** If the Series is empty.

See also:

[`Series.idxmax`](#) Return index *label* of the first occurrence of maximum of values.

**Notes**

This method is the Series version of `ndarray.argmax`. This method returns the label of the minimum, while `ndarray.argmax` returns the position. To get the position, use `series.values.argmax()`.

**Examples**

```
>>> s = ks.Series(data=[1, None, 4, 0],
...               index=['A', 'B', 'C', 'D'])
>>> s
A    1.0
B    NaN
C    4.0
D    0.0
dtype: float64
```

```
>>> s.idxmin()
'D'
```

If *skipna* is False and there is an NA value in the data, the function returns nan.

```
>>> s.idxmin(skipna=False)
nan
```

In case of multi-index, you get a tuple:

```
>>> index = pd.MultiIndex.from_arrays([
...     ['a', 'a', 'b', 'b'], ['c', 'd', 'e', 'f']], names=('first', 'second'))
>>> s = ks.Series(data=[1, None, 4, 0], index=index)
>>> s
first second
a      c      1.0
      d      NaN
```

(continues on next page)

(continued from previous page)

```
b      e      4.0
      f      0.0
dtype: float64
```

```
>>> s.idxmin()
('b', 'f')
```

If multiple values equal the minimum, the first row label with that value is returned.

```
>>> s = ks.Series([1, 100, 1, 100, 1, 100], index=[10, 3, 5, 2, 1, 8])
>>> s
10      1
3      100
5      1
2      100
1      1
8      100
dtype: int64
```

```
>>> s.idxmin()
10
```

### databricks.koalas.Series.isin

`Series.isin(values)` → Union[databricks.koalas.series.Series, databricks.koalas.indexes.base.Index]

Check whether *values* are contained in Series or Index.

Return a boolean Series or Index showing whether each element in the Series matches an element in the passed sequence of *values* exactly.

#### Parameters

**values** [set or list-like] The sequence of values to test.

#### Returns

**isin** [Series (bool dtype) or Index (bool dtype)]

### Examples

```
>>> s = ks.Series(['lama', 'cow', 'lama', 'beetle', 'lama',
...               'hippo'], name='animal')
>>> s.isin(['cow', 'lama'])
0      True
1      True
2      True
3     False
4      True
5     False
Name: animal, dtype: bool
```

Passing a single string as `s.isin('lama')` will raise an error. Use a list of one element instead:

```
>>> s.isin(['lama'])
0      True
1     False
2      True
3     False
4      True
5     False
Name: animal, dtype: bool
```

```
>>> s.rename("a").to_frame().set_index("a").index.isin(['lama'])
Index([True, False, True, False, True, False], dtype='object', name='a')
```

## **databricks.koalas.Series.last**

`Series.last` (*offset: Union[str, pandas.\_libs.tslibs.offsets.DateOffset]*) → `databricks.koalas.series.Series`  
Select final periods of time series data based on a date offset.

When having a Series with dates as index, this function can select the last few elements based on a date offset.

### **Parameters**

**offset** [str or `DateOffset`] The offset length of the data that will be selected. For instance, ‘3D’ will display all the rows having their index within the last 3 days.

### **Returns**

**Series** A subset of the caller.

### **Raises**

**TypeError** If the index is not a *DatetimeIndex*

## **Examples**

```
>>> index = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> kser = ks.Series([1, 2, 3, 4], index=index)
>>> kser
2018-04-09    1
2018-04-11    2
2018-04-13    3
2018-04-15    4
dtype: int64
```

Get the rows for the last 3 days:

```
>>> kser.last('3D')
2018-04-13    3
2018-04-15    4
dtype: int64
```

Notice the data for 3 last calendar days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

**databricks.koalas.Series.rename**

`Series.rename(index=None, **kwargs) → databricks.koalas.series.Series`  
 Alter Series name.

**Parameters**

**index** [scalar] Scalar will alter the `Series.name` attribute.

**inplace** [bool, default False] Whether to return a new Series. If True then value of copy is ignored.

**Returns**

**Series** Series with name altered.

**Examples**

```
>>> s = ks.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.rename("my_name") # scalar, changes Series.name
0    1
1    2
2    3
Name: my_name, dtype: int64
```

**databricks.koalas.Series.rename\_axis**

`Series.rename_axis mapper: Optional[Any] = None, index: Optional[Any] = None, inplace: bool = False) → Optional[databricks.koalas.series.Series]`  
 Set the name of the axis for the index or columns.

**Parameters**

**mapper, index** [scalar, list-like, dict-like or function, optional] A scalar, list-like, dict-like or functions transformations to apply to the index values.

**inplace** [bool, default False] Modifies the object directly, instead of creating a new Series.

**Returns**

**Series, or None if *inplace* is True.**

See also:

[\*Series.rename\*](#) Alter Series index labels or name.

[\*DataFrame.rename\*](#) Alter DataFrame index labels or name.

[\*Index.rename\*](#) Set new names on index.

## Examples

```
>>> s = ks.Series(["dog", "cat", "monkey"], name="animal")
>>> s
0      dog
1      cat
2    monkey
Name: animal, dtype: object
>>> s.rename_axis("index").sort_index()
index
0      dog
1      cat
2    monkey
Name: animal, dtype: object
```

## MultiIndex

```
>>> index = pd.MultiIndex.from_product([['mammal'],
...                                     ['dog', 'cat', 'monkey']],
...                                     names=['type', 'name'])
>>> s = ks.Series([4, 4, 2], index=index, name='num_legs')
>>> s
type      name      num_legs
mammal    dog         4
          cat         4
          monkey      2
Name: num_legs, dtype: int64
>>> s.rename_axis(index={'type': 'class'}).sort_index()
class      name      num_legs
mammal    cat         4
          dog         4
          monkey      2
Name: num_legs, dtype: int64
>>> s.rename_axis(index=str.upper).sort_index()
TYPE      NAME      num_legs
mammal    cat         4
          dog         4
          monkey      2
Name: num_legs, dtype: int64
```

## databricks.koalas.Series.reindex

`Series.reindex(index: Optional[Any] = None, fill_value: Optional[Any] = None) →`

`databricks.koalas.series.Series`

Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced.

### Parameters

**index:** array-like, optional New labels / index to conform to, should be specified using keywords. Preferably an Index object to avoid duplicating data

**fill\_value** [scalar, default np.NaN] Value to use for missing values. Defaults to NaN, but can be any “compatible” value.

### Returns

Series with changed index.



See also:

`Series.reset_index` Remove row labels or move them to new columns.

## Examples

Create a series with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> ser = ks.Series([200, 200, 404, 404, 301],
...                  index=index, name='http_status')
>>> ser
Firefox      200
Chrome       200
Safari       404
IE10         404
Konqueror    301
Name: http_status, dtype: int64
```

Create a new index and reindex the Series. By default values in the new index that do not have corresponding records in the Series are assigned NaN.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> ser.reindex(new_index).sort_index()
Chrome      200.0
Comodo Dragon  NaN
IE10        404.0
Iceweasel   NaN
Safari      404.0
Name: http_status, dtype: float64
```

We can fill in the missing values by passing a value to the keyword `fill_value`.

```
>>> ser.reindex(new_index, fill_value=0).sort_index()
Chrome      200
Comodo Dragon  0
IE10        404
Iceweasel    0
Safari      404
Name: http_status, dtype: int64
```

To further illustrate the filling functionality in `reindex`, we will create a Series with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> ser2 = ks.Series([100, 101, np.nan, 100, 89, 88],
...                  name='prices', index=date_index)
>>> ser2.sort_index()
2010-01-01    100.0
2010-01-02    101.0
2010-01-03     NaN
2010-01-04    100.0
2010-01-05     89.0
2010-01-06     88.0
Name: prices, dtype: float64
```

Suppose we decide to expand the series to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> ser2.reindex(date_index2).sort_index()
2009-12-29      NaN
2009-12-30      NaN
2009-12-31      NaN
2010-01-01    100.0
2010-01-02    101.0
2010-01-03      NaN
2010-01-04    100.0
2010-01-05     89.0
2010-01-06     88.0
2010-01-07      NaN
Name: prices, dtype: float64
```

### `databricks.koalas.Series.reindex_like`

`Series.reindex_like` (*other: Union[Series, DataFrame]*) → `databricks.koalas.series.Series`

Return a Series with matching indices as other object.

Conform the object to the same index on all axes. Places NA/NaN in locations having no value in the previous index.

#### Parameters

**other** [Series or DataFrame] Its row and column indices are used to define the new indices of this object.

#### Returns

**Series** Series with changed indices on each axis.

See also:

`DataFrame.set_index` Set row labels.

`DataFrame.reset_index` Remove row labels or move them to new columns.

`DataFrame.reindex` Change to new indices or expand indices.

### Notes

Same as calling `.reindex(index=other.index, ...)`.

### Examples

```
>>> s1 = ks.Series([24.3, 31.0, 22.0, 35.0],
...                 index=pd.date_range(start='2014-02-12',
...                                     end='2014-02-15', freq='D'),
...                 name="temp_celsius")
>>> s1
2014-02-12    24.3
2014-02-13    31.0
2014-02-14    22.0
2014-02-15    35.0
Name: temp_celsius, dtype: float64
```

```
>>> s2 = ks.Series(["low", "low", "medium"],
...                 index=pd.DatetimeIndex(['2014-02-12', '2014-02-13',
...                                         '2014-02-15']),
...                 name="winspeed")
>>> s2
2014-02-12    low
2014-02-13    low
2014-02-15  medium
Name: winspeed, dtype: object
```

```
>>> s2.reindex_like(s1).sort_index()
2014-02-12    low
2014-02-13    low
2014-02-14    None
2014-02-15  medium
Name: winspeed, dtype: object
```

### databricks.koalas.Series.reset\_index

`Series.reset_index(level=None, drop=False, name=None, inplace=False)` →  
 Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame, None]  
 Generate a new DataFrame or Series with the index reset.

This is useful when the index needs to be treated as a column, or when the index is meaningless and needs to be reset to the default before another operation.

#### Parameters

**level** [int, str, tuple, or list, default optional] For a Series with a MultiIndex, only remove the specified levels from the index. Removes all levels by default.

**drop** [bool, default False] Just reset the index, without inserting it as a column in the new DataFrame.

**name** [object, optional] The name to use for the column containing the original Series values. Uses self.name by default. This argument is ignored when drop is True.

**inplace** [bool, default False] Modify the Series in place (do not create a new object).

#### Returns

**Series or DataFrame** When *drop* is False (the default), a DataFrame is returned. The newly created columns will come first in the DataFrame, followed by the original Series values. When *drop* is True, a *Series* is returned. In either case, if *inplace*=True, no value is returned.

### Examples

```
>>> s = ks.Series([1, 2, 3, 4], index=pd.Index(['a', 'b', 'c', 'd'], name='idx'))
```

Generate a DataFrame with default index.

```
>>> s.reset_index()
   idx  0
0    a  1
1    b  2
```

(continues on next page)

(continued from previous page)

```
2  c  3
3  d  4
```

To specify the name of the new column use *name*.

```
>>> s.reset_index(name='values')
   idx  values
0    a      1
1    b      2
2    c      3
3    d      4
```

To generate a new Series with the default set *drop* to True.

```
>>> s.reset_index(drop=True)
0    1
1    2
2    3
3    4
dtype: int64
```

To update the Series in place, without generating a new one set *inplace* to True. Note that it also requires *drop=True*.

```
>>> s.reset_index(inplace=True, drop=True)
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

## databricks.koalas.Series.sample

`Series.sample(n: Optional[int] = None, frac: Optional[float] = None, replace: bool = False, random_state: Optional[int] = None) → databricks.koalas.series.Series`

Return a random sample of items from an axis of object.

Please call this function using named argument by specifying the *frac* argument.

You can use *random\_state* for reproducibility. However, note that different from pandas, specifying a seed in Koalas/Spark does not guarantee the sampled rows will be fixed. The result set depends on not only the seed, but also how the data is distributed across machines and to some extent network randomness when shuffle operations are involved. Even in the simplest case, the result set will depend on the system's CPU core count.

### Parameters

**n** [int, optional] Number of items to return. This is currently NOT supported. Use *frac* instead.

**frac** [float, optional] Fraction of axis items to return.

**replace** [bool, default False] Sample with or without replacement.

**random\_state** [int, optional] Seed for the random number generator (if int).

### Returns

**Series or DataFrame** A new object of same type as caller containing the sampled items.

## Examples

```
>>> df = ks.DataFrame({'num_legs': [2, 4, 8, 0],
...                     'num_wings': [2, 0, 0, 0],
...                     'num_specimen_seen': [10, 2, 1, 8]},
...                     index=['falcon', 'dog', 'spider', 'fish'],
...                     columns=['num_legs', 'num_wings', 'num_specimen_seen'])
>>> df
```

	num_legs	num_wings	num_specimen_seen
falcon	2	2	10
dog	4	0	2
spider	8	0	1
fish	0	0	8

A random 25% sample of the DataFrame. Note that we use *random\_state* to ensure the reproducibility of the examples.

```
>>> df.sample(frac=0.25, random_state=1)
```

	num_legs	num_wings	num_specimen_seen
falcon	2	2	10
fish	0	0	8

Extract 25% random elements from the Series `df['num_legs']`, with replacement, so the same items could appear more than once.

```
>>> df['num_legs'].sample(frac=0.4, replace=True, random_state=1)
```

falcon	2
spider	8
spider	8

Name: num\_legs, dtype: int64

Specifying the exact number of items to return is not supported at the moment.

```
>>> df.sample(n=5)
Traceback (most recent call last):
...
NotImplementedError: Function sample currently does not support specifying ...
```

## databricks.koalas.Series.swaplevel

`Series.swaplevel(i=-2, j=-1, copy: bool = True) → databricks.koalas.series.Series`  
 Swap levels *i* and *j* in a MultiIndex. Default is to swap the two innermost levels of the index.

### Parameters

**i, j** [int, str] Level of the indices to be swapped. Can pass level name as string.

**copy** [bool, default True] Whether to copy underlying data. Must be True.

### Returns

**Series** Series with levels swapped in MultiIndex.

## Examples

```
>>> midx = pd.MultiIndex.from_arrays(['a', 'b'], [1, 2], names = ['word',
↳ 'number'])
>>> midx
MultiIndex([('a', 1),
            ('b', 2)],
            names=['word', 'number'])
>>> kser = ks.Series(['x', 'y'], index=midx)
>>> kser
word  number
a      1      x
b      2      y
dtype: object
>>> kser.swaplevel()
number  word
1      a      x
2      b      y
dtype: object
>>> kser.swaplevel(0, 1)
number  word
1      a      x
2      b      y
dtype: object
>>> kser.swaplevel('number', 'word')
number  word
1      a      x
2      b      y
dtype: object
```

## databricks.koalas.Series.swapaxes

`Series.swapaxes` (*i*: `Union[str, int]`, *j*: `Union[str, int]`, *copy*: `bool = True`) → `databricks.koalas.series.Series`  
Interchange axes and swap values axes appropriately.

### Parameters

**i**: {0 or 'index', 1 or 'columns'}. The axis to swap.

**j**: {0 or 'index', 1 or 'columns'}. The axis to swap.

**copy** [bool, default True.]

### Returns

**Series**

## Examples

```
>>> kser = ks.Series([1, 2, 3], index=["x", "y", "z"])
>>> kser
x    1
y    2
z    3
dtype: int64
>>>
>>> kser.swapaxes(0, 0)
x    1
y    2
z    3
dtype: int64
```

## databricks.koalas.Series.take

`Series.take(indices) → Union[databricks.koalas.series.Series, databricks.koalas.indexes.base.Index]`

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

### Parameters

**indices** [array-like] An array of ints indicating which positions to take.

### Returns

**taken** [same type as caller] An array-like containing the elements taken from the object.

See also:

[`DataFrame.loc`](#) Select a subset of a DataFrame by labels.

[`DataFrame.iloc`](#) Select a subset of a DataFrame by positions.

[`numpy.take`](#) Take elements from an array along an axis.

## Examples

Series

```
>>> kser = ks.Series([100, 200, 300, 400, 500])
>>> kser
0    100
1    200
2    300
3    400
4    500
dtype: int64
```

```
>>> kser.take([0, 2, 4]).sort_index()
0    100
2    300
4    500
dtype: int64
```

### Index

```
>>> kidx = ks.Index([100, 200, 300, 400, 500])
>>> kidx
Int64Index([100, 200, 300, 400, 500], dtype='int64')
```

```
>>> kidx.take([0, 2, 4]).sort_values()
Int64Index([100, 300, 500], dtype='int64')
```

### MultiIndex

```
>>> kmidx = ks.MultiIndex.from_tuples([("x", "a"), ("x", "b"), ("x", "c")])
>>> kmidx
MultiIndex([('x', 'a'),
            ('x', 'b'),
            ('x', 'c')],
           )
```

```
>>> kmidx.take([0, 2])
MultiIndex([('x', 'a'),
            ('x', 'c')],
           )
```

## databricks.koalas.Series.tail

`Series.tail (n=5)` → `databricks.koalas.series.Series`

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

For negative values of *n*, this function returns all rows except the first *n* rows, equivalent to `df[n:]`.

#### Parameters

**n** [int, default 5] Number of rows to select.

#### Returns

**type of caller** The last *n* rows of the caller object.

**See also:**

**`DataFrame.head`** The first *n* rows of the caller object.

### Examples

```
>>> kser = ks.Series([1, 2, 3, 4, 5])
>>> kser
0      1
1      2
2      3
3      4
4      5
dtype: int64
```



```
>>> kser.tail(3)
2      3
3      4
4      5
dtype: int64
```

### `databricks.koalas.Series.where`

`Series.where(cond, other=nan) → databricks.koalas.series.Series`

Replace values where the condition is False.

#### Parameters

**cond** [boolean Series] Where cond is True, keep the original value. Where False, replace with corresponding value from other.

**other** [scalar, Series] Entries where cond is False are replaced with corresponding value from other.

#### Returns

**Series**

### Examples

```
>>> from databricks.koalas.config import set_option, reset_option
>>> set_option("compute.ops_on_diff_frames", True)
>>> s1 = ks.Series([0, 1, 2, 3, 4])
>>> s2 = ks.Series([100, 200, 300, 400, 500])
>>> s1.where(s1 > 0).sort_index()
0      NaN
1      1.0
2      2.0
3      3.0
4      4.0
dtype: float64
```

```
>>> s1.where(s1 > 1, 10).sort_index()
0      10
1      10
2       2
3       3
4       4
dtype: int64
```

```
>>> s1.where(s1 > 1, s1 + 100).sort_index()
0      100
1      101
2       2
3       3
4       4
dtype: int64
```

```
>>> s1.where(s1 > 1, s2).sort_index()
0      100
1      200
2         2
3         3
4         4
dtype: int64
```

```
>>> reset_option("compute.ops_on_diff_frames")
```

### `databricks.koalas.Series.mask`

`Series.mask(cond, other=nan) → databricks.koalas.series.Series`

Replace values where the condition is True.

#### Parameters

**cond** [boolean Series] Where cond is False, keep the original value. Where True, replace with corresponding value from other.

**other** [scalar, Series] Entries where cond is True are replaced with corresponding value from other.

#### Returns

**Series**

### Examples

```
>>> from databricks.koalas.config import set_option, reset_option
>>> set_option("compute.ops_on_diff_frames", True)
>>> s1 = ks.Series([0, 1, 2, 3, 4])
>>> s2 = ks.Series([100, 200, 300, 400, 500])
>>> s1.mask(s1 > 0).sort_index()
0      0.0
1      NaN
2      NaN
3      NaN
4      NaN
dtype: float64
```

```
>>> s1.mask(s1 > 1, 10).sort_index()
0      0
1      1
2     10
3     10
4     10
dtype: int64
```

```
>>> s1.mask(s1 > 1, s1 + 100).sort_index()
0      0
1      1
2     102
3     103
```

(continues on next page)

(continued from previous page)

```
4      104
dtype: int64
```

```
>>> s1.mask(s1 > 1, s2).sort_index()
0      0
1      1
2     300
3     400
4     500
dtype: int64
```

```
>>> reset_option("compute.ops_on_diff_frames")
```

### databricks.koalas.Series.truncate

`Series.truncate` (*before=None, after=None, axis=None, copy=True*) → Union[DataFrame, Series]

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

---

**Note:** This API is dependent on `Index.is_monotonic_increasing()` which can be expensive.

---

#### Parameters

**before** [date, str, int] Truncate all rows before this index value.

**after** [date, str, int] Truncate all rows after this index value.

**axis** [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.

**copy** [bool, default is True,] Return a copy of the truncated section.

#### Returns

**type of caller** The truncated Series or DataFrame.

See also:

`DataFrame.loc` Select a subset of a DataFrame by label.

`DataFrame.iloc` Select a subset of a DataFrame by position.

### Examples

```
>>> df = ks.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
   A  B  C
1  a  f  k
2  b  g  l
```

(continues on next page)

(continued from previous page)

```
3 c h m
4 d i n
5 e j o
```

```
>>> df.truncate(before=2, after=4)
   A B C
2 b g l
3 c h m
4 d i n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A B
1 a f
2 b g
3 c h
4 d i
5 e j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

A Series has index that sorted integers.

```
>>> s = ks.Series([10, 20, 30, 40, 50, 60, 70],
...               index=[1, 2, 3, 4, 5, 6, 7])
>>> s
1    10
2    20
3    30
4    40
5    50
6    60
7    70
dtype: int64
```

```
>>> s.truncate(2, 5)
2    20
3    30
4    40
5    50
dtype: int64
```

A Series has index that sorted strings.

```
>>> s = ks.Series([10, 20, 30, 40, 50, 60, 70],
...               index=['a', 'b', 'c', 'd', 'e', 'f', 'g'])
>>> s
a    10
b    20
```

(continues on next page)

(continued from previous page)

```
c    30
d    40
e    50
f    60
g    70
dtype: int64
```

```
>>> s.truncate('b', 'e')
b    20
c    30
d    40
e    50
dtype: int64
```

### 3.3.9 Missing data handling

<code>Series.backfill([axis, inplace, limit])</code>	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='bfill'</code> .
<code>Series.bfill([axis, inplace, limit])</code>	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='bfill'</code> .
<code>Series.isna()</code>	Detect existing (non-missing) values.
<code>Series.isnull()</code>	Detect existing (non-missing) values.
<code>Series.notna()</code>	Detect existing (non-missing) values.
<code>Series.notnull()</code>	Detect existing (non-missing) values.
<code>Series.pad([axis, inplace, limit])</code>	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='ffill'</code> .
<code>Series.dropna([axis, inplace])</code>	Return a new Series with missing values removed.
<code>Series.fillna([value, method, axis, ...])</code>	Fill NA/NaN values.

#### databricks.koalas.Series.backfill

`Series.backfill` (*axis=None, inplace=False, limit=None*) → Union[DataFrame, Series]

Synonym for `DataFrame.fillna()` or `Series.fillna()` with `method='bfill'`.

**Note:** the current implementation of 'bfill' uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

#### Parameters

**axis** [{0 or *index*}] 1 and *columns* are not supported.

**inplace** [boolean, default False] Fill in place (do not create a new object)

**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

## Returns

**DataFrame or Series** DataFrame or Series with NA entries filled.

## Examples

```
>>> kdf = ks.DataFrame({
...     'A': [None, 3, None, None],
...     'B': [2, 4, None, 3],
...     'C': [None, None, None, 1],
...     'D': [0, 1, 5, 4]
... },
... columns=['A', 'B', 'C', 'D'])
>>> kdf
```

	A	B	C	D
0	NaN	2.0	NaN	0
1	3.0	4.0	NaN	1
2	NaN	NaN	NaN	5
3	NaN	3.0	1.0	4

Propagate non-null values backward.

```
>>> kdf.bfill()
```

	A	B	C	D
0	3.0	2.0	1.0	0
1	3.0	4.0	1.0	1
2	NaN	3.0	1.0	5
3	NaN	3.0	1.0	4

## For Series

```
>>> kser = ks.Series([None, None, None, 1])
>>> kser
```

0	NaN
1	NaN
2	NaN
3	1.0

dtype: float64

```
>>> kser.bfill()
```

0	1.0
1	1.0
2	1.0
3	1.0

dtype: float64

**databricks.koalas.Series.bfill**

`Series.bfill` (*axis=None, inplace=False, limit=None*) → Union[DataFrame, Series]  
 Synonym for `DataFrame.fillna()` or `Series.fillna()` with `method='bfill'`.

**Note:** the current implementation of 'bfill' uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

**Parameters**

**axis** [[0 or *index*]] 1 and *columns* are not supported.

**inplace** [boolean, default False] Fill in place (do not create a new object)

**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

**Returns**

**DataFrame or Series** DataFrame or Series with NA entries filled.

**Examples**

```
>>> kdf = ks.DataFrame({
...     'A': [None, 3, None, None],
...     'B': [2, 4, None, 3],
...     'C': [None, None, None, 1],
...     'D': [0, 1, 5, 4]
... },
... columns=['A', 'B', 'C', 'D'])
>>> kdf
```

	A	B	C	D
0	NaN	2.0	NaN	0
1	3.0	4.0	NaN	1
2	NaN	NaN	NaN	5
3	NaN	3.0	1.0	4

Propagate non-null values backward.

```
>>> kdf.bfill()
```

	A	B	C	D
0	3.0	2.0	1.0	0
1	3.0	4.0	1.0	1
2	NaN	3.0	1.0	5
3	NaN	3.0	1.0	4

For Series

```
>>> kser = ks.Series([None, None, None, 1])
>>> kser
```

0	NaN
---	-----

(continues on next page)

(continued from previous page)

```
1    NaN
2    NaN
3    1.0
dtype: float64
```

```
>>> kser.bfill()
0    1.0
1    1.0
2    1.0
3    1.0
dtype: float64
```

### databricks.koalas.Series.isna

`Series.isna()` → Union[databricks.koalas.series.Series, databricks.koalas.indexes.base.Index]

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy.NaN, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use\_inf\_as\_na = True).

#### Returns

**Series or Index** [Mask of bool values for each element in Series] that indicates whether an element is not an NA value.

### Examples

```
>>> ser = ks.Series([5, 6, np.NaN])
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

```
>>> ser.rename("a").to_frame().set_index("a").index.isna()
Index([False, False, True], dtype='object', name='a')
```

### databricks.koalas.Series.isnull

`Series.isnull()` → Union[databricks.koalas.series.Series, databricks.koalas.indexes.base.Index]

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy.NaN, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use\_inf\_as\_na = True).

#### Returns

**Series or Index** [Mask of bool values for each element in Series] that indicates whether an element is not an NA value.



## Examples

```
>>> ser = ks.Series([5, 6, np.NaN])
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

```
>>> ser.rename("a").to_frame().set_index("a").index.isna()
Index([False, False, True], dtype='object', name='a')
```

## databricks.koalas.Series.notna

`Series.notna()` → Union[databricks.koalas.series.Series, databricks.koalas.indexes.base.Index]

Detect existing (non-missing) values. Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings “” or numpy.inf are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or numpy.NaN, get mapped to False values.

### Returns

**Series or Index** [Mask of bool values for each element in Series] that indicates whether an element is not an NA value.

## Examples

Show which entries in a Series are not NA.

```
>>> ser = ks.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

```
>>> ser.rename("a").to_frame().set_index("a").index.notna()
Index([True, True, False], dtype='object', name='a')
```

### `databricks.koalas.Series.notnull`

`Series.notnull()` → `Union[databricks.koalas.series.Series, databricks.koalas.indexes.base.Index]`

Detect existing (non-missing) values. Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as `None` or `numpy.NaN`, get mapped to False values.

#### Returns

**Series or Index** [Mask of bool values for each element in Series] that indicates whether an element is not an NA value.

### Examples

Show which entries in a Series are not NA.

```
>>> ser = ks.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

```
>>> ser.rename("a").to_frame().set_index("a").index.notna()
Index([True, True, False], dtype='object', name='a')
```

### `databricks.koalas.Series.pad`

`Series.pad(axis=None, inplace=False, limit=None)` → `Union[DataFrame, Series]`

Synonym for `DataFrame.fillna()` or `Series.fillna()` with `method='ffill'`.

---

**Note:** the current implementation of 'ffill' uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

#### Parameters

**axis** [{0 or index}] 1 and columns are not supported.

**inplace** [boolean, default False] Fill in place (do not create a new object)

**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

**Returns****DataFrame or Series** DataFrame or Series with NA entries filled.**Examples**

```
>>> kdf = ks.DataFrame({
...     'A': [None, 3, None, None],
...     'B': [2, 4, None, 3],
...     'C': [None, None, None, 1],
...     'D': [0, 1, 5, 4]
...     },
...     columns=['A', 'B', 'C', 'D'])
>>> kdf
```

	A	B	C	D
0	NaN	2.0	NaN	0
1	3.0	4.0	NaN	1
2	NaN	NaN	NaN	5
3	NaN	3.0	1.0	4

Propagate non-null values forward.

```
>>> kdf.ffmpeg()
A      B      C      D
0  NaN  2.0  NaN  0
1  3.0  4.0  NaN  1
2  3.0  4.0  NaN  5
3  3.0  3.0  1.0  4
```

**For Series**

```
>>> kser = ks.Series([2, 4, None, 3])
>>> kser
0      2.0
1      4.0
2      NaN
3      3.0
dtype: float64
```

```
>>> kser.ffmpeg()
0      2.0
1      4.0
2      4.0
3      3.0
dtype: float64
```

**databricks.koalas.Series.dropna**

`Series.dropna` (*axis=0*, *inplace=False*, *\*\*kwargs*) → `Optional[databricks.koalas.series.Series]`  
Return a new Series with missing values removed.

**Parameters**

**axis** [{0 or 'index'}, default 0] There is only one axis to drop values from.

**inplace** [bool, default False] If True, do operation inplace and return None.

**\*\*kwargs** Not in use.

**Returns**

**Series** Series with NA entries dropped from it.

**Examples**

```
>>> ser = ks.Series([1., 2., np.nan])
>>> ser
0    1.0
1    2.0
2    NaN
dtype: float64
```

Drop NA values from a Series.

```
>>> ser.dropna()
0    1.0
1    2.0
dtype: float64
```

Keep the Series with valid entries in the same variable.

```
>>> ser.dropna(inplace=True)
>>> ser
0    1.0
1    2.0
dtype: float64
```

**databricks.koalas.Series.fillna**

`Series.fillna` (*value=None*, *method=None*, *axis=None*, *inplace=False*, *limit=None*) → `Optional[databricks.koalas.series.Series]`  
Fill NA/NaN values.

---

**Note:** the current implementation of ‘method’ parameter in `fillna` uses Spark’s Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

**Parameters**

**value** [scalar, dict, Series] Value to use to fill holes. alternately a dict/Series of values specifying which value to use for each column. DataFrame is not supported.

**method** [{ 'backfill', 'bfill', 'pad', 'ffill', None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

**axis** [{0 or *index*}] 1 and *columns* are not supported.

**inplace** [boolean, default False] Fill in place (do not create a new object)

**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

### Returns

**Series** Series with NA entries filled.

### Examples

```
>>> s = ks.Series([np.nan, 2, 3, 4, np.nan, 6], name='x')
>>> s
0    NaN
1    2.0
2    3.0
3    4.0
4    NaN
5    6.0
Name: x, dtype: float64
```

Replace all NaN elements with 0s.

```
>>> s.fillna(0)
0    0.0
1    2.0
2    3.0
3    4.0
4    0.0
5    6.0
Name: x, dtype: float64
```

We can also propagate non-null values forward or backward.

```
>>> s.fillna(method='ffill')
0    NaN
1    2.0
2    3.0
3    4.0
4    4.0
5    6.0
Name: x, dtype: float64
```

```
>>> s = ks.Series([np.nan, 'a', 'b', 'c', np.nan], name='x')
>>> s.fillna(method='ffill')
0    None
1     a
2     b
```

(continues on next page)

(continued from previous page)

```

3      c
4      c
Name: x, dtype: object

```

### 3.3.10 Reshaping, sorting, transposing

<code>Series.argsort()</code>	Return the integer indices that would sort the Series values.
<code>Series.argmin()</code>	Return int position of the smallest value in the Series.
<code>Series.argmax()</code>	Return int position of the largest value in the Series.
<code>Series.sort_index([axis, level, ascending, ...])</code>	Sort object by labels (along an axis)
<code>Series.sort_values([ascending, inplace, ...])</code>	Sort by the values.
<code>Series.unstack([level])</code>	Unstack, a.k.a.
<code>Series.explode()</code>	Transform each element of a list-like to a row.
<code>Series.repeat(repeats)</code>	Repeat elements of a Series.
<code>Series.squeeze([axis])</code>	Squeeze 1 dimensional axis objects into scalars.
<code>Series.factorize([sort, na_sentinel])</code>	Encode the object as an enumerated type or categorical variable.

#### databricks.koalas.Series.argsort

`Series.argsort()` → `databricks.koalas.series.Series`

Return the integer indices that would sort the Series values. Unlike pandas, the index order is not preserved in the result.

#### Returns

**Series** Positions of values within the sort order with -1 indicating nan values.

#### Examples

```

>>> kser = ks.Series([3, 3, 4, 1, 6, 2, 3, 7, 8, 7, 10])
>>> kser
0      3
1      3
2      4
3      1
4      6
5      2
6      3
7      7
8      8
9      7
10     10
dtype: int64

```

```

>>> kser.argsort().sort_index()
0      3
1      5
2      0

```

(continues on next page)

(continued from previous page)

```

3      1
4      6
5      2
6      4
7      7
8      9
9      8
10     10
dtype: int64

```

**databricks.koalas.Series.argmax****Series.argmax()** → int

Return int position of the smallest value in the Series.

If the minimum is achieved in multiple locations, the first row position is returned.

**Returns****int** Row position of the minimum value.**Examples**

Consider dataset containing cereal calories

```

>>> s = ks.Series({'Corn Flakes': 100.0, 'Almond Delight': 110.0,
...                'Cinnamon Toast Crunch': 120.0, 'Cocoa Puff': 110.0})
>>> s
Corn Flakes          100.0
Almond Delight       110.0
Cinnamon Toast Crunch 120.0
Cocoa Puff           110.0
dtype: float64

```

```

>>> s.argmax()
0

```

**databricks.koalas.Series.argmax****Series.argmax()** → int

Return int position of the largest value in the Series.

If the maximum is achieved in multiple locations, the first row position is returned.

**Returns****int** Row position of the maximum value.

## Examples

Consider dataset containing cereal calories

```
>>> s = ks.Series({'Corn Flakes': 100.0, 'Almond Delight': 110.0,
...               'Cinnamon Toast Crunch': 120.0, 'Cocoa Puff': 110.0})
>>> s
Corn Flakes          100.0
Almond Delight       110.0
Cinnamon Toast Crunch 120.0
Cocoa Puff           110.0
dtype: float64
```

```
>>> s.argmax()
2
```

## databricks.koalas.Series.sort\_index

**Series.sort\_index** (*axis*: int = 0, *level*: Union[int, List[int], None] = None, *ascending*: bool = True, *inplace*: bool = False, *kind*: str = None, *na\_position*: str = 'last') → Optional[databricks.koalas.series.Series]  
Sort object by labels (along an axis)

### Parameters

- axis** [index, columns to direct sorting. Currently, only axis = 0 is supported.]
- level** [int or level name or list of ints or list of level names] if not None, sort on values in specified index level(s)
- ascending** [boolean, default True] Sort ascending vs. descending
- inplace** [bool, default False] if True, perform operation in-place
- kind** [str, default None] Koalas does not allow specifying the sorting algorithm at the moment, default None
- na\_position** [{‘first’, ‘last’}, default ‘last’] first puts NaNs at the beginning, last puts NaNs at the end. Not implemented for MultiIndex.

### Returns

**sorted\_obj** [Series]

## Examples

```
>>> df = ks.Series([2, 1, np.nan], index=['b', 'a', np.nan])
```

```
>>> df.sort_index()
a      1.0
b      2.0
NaN    NaN
dtype: float64
```



```
>>> df.sort_index(ascending=False)
b      2.0
a      1.0
NaN     NaN
dtype: float64
```

```
>>> df.sort_index(na_position='first')
NaN     NaN
a      1.0
b      2.0
dtype: float64
```

```
>>> df.sort_index(inplace=True)
>>> df
a      1.0
b      2.0
NaN     NaN
dtype: float64
```

```
>>> df = ks.Series(range(4), index=[['b', 'b', 'a', 'a'], [1, 0, 1, 0]], name='0')
```

```
>>> df.sort_index()
a  0    3
   1    2
b  0    1
   1    0
Name: 0, dtype: int64
```

```
>>> df.sort_index(level=1)
a  0    3
b  0    1
a  1    2
b  1    0
Name: 0, dtype: int64
```

```
>>> df.sort_index(level=[1, 0])
a  0    3
b  0    1
a  1    2
b  1    0
Name: 0, dtype: int64
```

### **databricks.koalas.Series.sort\_values**

`Series.sort_values` (*ascending: bool = True, inplace: bool = False, na\_position: str = 'last'*) → Optional[databricks.koalas.series.Series]

Sort by the values.

Sort a Series in ascending or descending order by some criterion.

#### **Parameters**

**ascending** [bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

**inplace** [bool, default False] if True, perform operation in-place

**na\_position** [{‘first’, ‘last’}, default ‘last’] *first* puts NaNs at the beginning, *last* puts NaNs at the end

### Returns

**sorted\_obj** [Series ordered by values.]

### Examples

```
>>> s = ks.Series([np.nan, 1, 3, 10, 5])
>>> s
0      NaN
1      1.0
2      3.0
3     10.0
4      5.0
dtype: float64
```

Sort values ascending order (default behaviour)

```
>>> s.sort_values(ascending=True)
1      1.0
2      3.0
4      5.0
3     10.0
0      NaN
dtype: float64
```

Sort values descending order

```
>>> s.sort_values(ascending=False)
3     10.0
4      5.0
2      3.0
1      1.0
0      NaN
dtype: float64
```

Sort values inplace

```
>>> s.sort_values(ascending=False, inplace=True)
>>> s
3     10.0
4      5.0
2      3.0
1      1.0
0      NaN
dtype: float64
```

Sort values putting NAs first

```
>>> s.sort_values(na_position='first')
0      NaN
1      1.0
2      3.0
4      5.0
3     10.0
dtype: float64
```

Sort a series of strings

```
>>> s = ks.Series(['z', 'b', 'd', 'a', 'c'])
>>> s
0      z
1      b
2      d
3      a
4      c
dtype: object
```

```
>>> s.sort_values()
3      a
1      b
4      c
2      d
0      z
dtype: object
```

### databricks.koalas.Series.unstack

`Series.unstack(level=-1) → databricks.koalas.frame.DataFrame`

Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame. The level involved will automatically get sorted.

#### Parameters

**level** [int, str, or list of these, default last level] Level(s) to unstack, can pass level name.

#### Returns

**DataFrame** Unstacked Series.

### Notes

Unlike pandas, Koalas doesn't check whether an index is duplicated or not because the checking of duplicated index requires scanning whole data which can be quite expensive.

### Examples

```
>>> s = ks.Series([1, 2, 3, 4],
...               index=pd.MultiIndex.from_product([['one', 'two'],
...         ['a', 'b']]))
>>> s
one  a    1
     b    2
two  a    3
     b    4
dtype: int64
```

```
>>> s.unstack(level=-1).sort_index()
      a  b
one  1  2
two  3  4
```

```
>>> s.unstack(level=0).sort_index()
      one  two
a      1    3
b      2    4
```

### **databricks.koalas.Series.explode**

`Series.explode()` → `databricks.koalas.series.Series`

Transform each element of a list-like to a row.

#### **Returns**

**Series** Exploded lists to rows; index will be duplicated for these rows.

#### **See also:**

***Series.str.split*** Split string values on specified separator.

***Series.unstack*** Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame.

***DataFrame.melt*** Unpivot a DataFrame from wide format to long format.

***DataFrame.explode*** Explode a DataFrame from list-like columns to long format.

### **Examples**

```
>>> kser = ks.Series([[1, 2, 3], [], [3, 4]])
>>> kser
0      [1, 2, 3]
1           []
2      [3, 4]
dtype: object
```

```
>>> kser.explode()
0      1.0
0      2.0
0      3.0
1      NaN
2      3.0
2      4.0
dtype: float64
```

### **databricks.koalas.Series.repeat**

`Series.repeat(repeats: Union[int, Series])` → `databricks.koalas.series.Series`

Repeat elements of a Series.

Returns a new Series where each element of the current Series is repeated consecutively a given number of times.

#### **Parameters**

**repeats** [int or Series] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty Series.

#### **Returns**

**Series** Newly created Series with repeated elements.

**See also:**

**`Index.repeat`** Equivalent function for Index.

### Examples

```
>>> s = ks.Series(['a', 'b', 'c'])
>>> s
0    a
1    b
2    c
dtype: object
>>> s.repeat(2)
0    a
1    b
2    c
0    a
1    b
2    c
dtype: object
>>> ks.Series([1, 2, 3]).repeat(0)
Series([], dtype: int64)
```

### **databricks.koalas.Series.squeeze**

**Series.squeeze** (*axis=None*) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, DataFrame, Series]

Squeeze 1 dimensional axis objects into scalars.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don't know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call *squeeze* to ensure you have a Series.

#### **Parameters**

**axis** [{0 or 'index', 1 or 'columns', None}, default None] A specific axis to squeeze. By default, all length-1 axes are squeezed.

#### **Returns**

**DataFrame, Series, or scalar** The projection after squeezing *axis* or all the axes.

**See also:**

**`Series.iloc`** Integer-location based indexing for selecting scalars.

**`DataFrame.iloc`** Integer-location based indexing for selecting Series.

**`Series.to_frame`** Inverse of DataFrame.squeeze for a single-column DataFrame.

## Examples

```
>>> primes = ks.Series([2, 3, 5, 7])
```

Slicing might produce a Series with a single value:

```
>>> even_primes = primes[primes % 2 == 0]
>>> even_primes
0    2
dtype: int64
```

```
>>> even_primes.squeeze()
2
```

Squeezing objects with more than one value in every axis does nothing:

```
>>> odd_primes = primes[primes % 2 == 1]
>>> odd_primes
1    3
2    5
3    7
dtype: int64
```

```
>>> odd_primes.squeeze()
1    3
2    5
3    7
dtype: int64
```

Squeezing is even more effective when used with DataFrames.

```
>>> df = ks.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
>>> df
   a  b
0  1  2
1  3  4
```

Slicing a single column will produce a DataFrame with the columns having only one value:

```
>>> df_a = df[['a']]
>>> df_a
   a
0  1
1  3
```

So the columns can be squeezed down, resulting in a Series:

```
>>> df_a.squeeze('columns')
0    1
1    3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```
>>> df_1a = df.loc[[1], ['a']]
>>> df_1a
```

(continues on next page)

(continued from previous page)

```
a
1  3
```

Squeezing the rows produces a single scalar Series:

```
>>> df_1a.squeeze('rows')
a      3
Name: 1, dtype: int64
```

Squeezing all axes will project directly into a scalar:

```
>>> df_1a.squeeze()
3
```

### databricks.koalas.Series.factorize

`Series.factorize` (*sort*: *bool* = *True*, *na\_sentinel*: *Optional[int]* = *-1*) → *Tuple[Union[databricks.koalas.series.Series, pandas.core.indexes.base.Index], databricks.koalas.indexes.base.Index]*

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values.

#### Parameters

**sort** [bool, default True]

**na\_sentinel** [int or None, default -1] Value to mark “not found”. If None, will not drop the NaN from the uniques of the values.

#### Returns

**codes** [Series or Index] A Series or Index that’s an indexer into *uniques*. *uniques.take(codes)* will have the same values as *values*.

**uniques** [pd.Index] The unique valid values.

---

**Note:** Even if there’s a missing value in *values*, *uniques* will *not* contain an entry for it.

---

### Examples

```
>>> kser = ks.Series(['b', None, 'a', 'c', 'b'])
>>> codes, uniques = kser.factorize()
>>> codes
0      1
1     -1
2      0
3      2
4      1
dtype: int32
>>> uniques
Index(['a', 'b', 'c'], dtype='object')
```

```
>>> codes, uniques = kser.factorize(na_sentinel=None)
>>> codes
0    1
1    3
2    0
3    2
4    1
dtype: int32
>>> uniques
Index(['a', 'b', 'c', None], dtype='object')
```

```
>>> codes, uniques = kser.factorize(na_sentinel=-2)
>>> codes
0    1
1   -2
2    0
3    2
4    1
dtype: int32
>>> uniques
Index(['a', 'b', 'c'], dtype='object')
```

For Index:

```
>>> kidx = ks.Index(['b', None, 'a', 'c', 'b'])
>>> codes, uniques = kidx.factorize()
>>> codes
Int64Index([1, -1, 0, 2, 1], dtype='int64')
>>> uniques
Index(['a', 'b', 'c'], dtype='object')
```

### 3.3.11 Combining / joining / merging

<code>Series.append(to_append[, ignore_index, ...])</code>	Concatenate two or more Series.
<code>Series.compare(other[, keep_shape, keep_equal])</code>	Compare to another Series and show the differences.
<code>Series.replace([to_replace, value, regex])</code>	Replace values given in <code>to_replace</code> with <code>value</code> .
<code>Series.update(other)</code>	Modify Series in place using non-NA values from passed Series.

#### databricks.koalas.Series.append

`Series.append(to_append: databricks.koalas.series.Series, ignore_index: bool = False, verify_integrity: bool = False) → databricks.koalas.series.Series`  
Concatenate two or more Series.

##### Parameters

**to\_append** [Series or list/tuple of Series]

**ignore\_index** [boolean, default False] If True, do not use the index labels.

**verify\_integrity** [boolean, default False] If True, raise Exception on creating index with duplicates

##### Returns



**appended** [Series]

### Examples

```
>>> s1 = ks.Series([1, 2, 3])
>>> s2 = ks.Series([4, 5, 6])
>>> s3 = ks.Series([4, 5, 6], index=[3,4,5])
```

```
>>> s1.append(s2)
0    1
1    2
2    3
0    4
1    5
2    6
dtype: int64
```

```
>>> s1.append(s3)
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

With `ignore_index` set to `True`:

```
>>> s1.append(s2, ignore_index=True)
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

### `databricks.koalas.Series.compare`

`Series.compare` (*other*: `databricks.koalas.series.Series`, *keep\_shape*: `bool = False`, *keep\_equal*: `bool = False`) → `databricks.koalas.frame.DataFrame`  
Compare to another Series and show the differences.

#### Parameters

**other** [Series] Object to compare with.

**keep\_shape** [bool, default False] If true, all rows and columns are kept. Otherwise, only the ones with different values are kept.

**keep\_equal** [bool, default False] If true, the result keeps values that are equal. Otherwise, equal values are shown as NaNs.

#### Returns

**DataFrame**

## Notes

Matching NaNs will not appear as a difference.

## Examples

```
>>> from databricks.koalas.config import set_option, reset_option
>>> set_option("compute.ops_on_diff_frames", True)
>>> s1 = ks.Series(["a", "b", "c", "d", "e"])
>>> s2 = ks.Series(["a", "a", "c", "b", "e"])
```

Align the differences on columns

```
>>> s1.compare(s2).sort_index()
   self other
1    b      a
3    d      b
```

Keep all original rows

```
>>> s1.compare(s2, keep_shape=True).sort_index()
   self other
0  None  None
1    b      a
2  None  None
3    d      b
4  None  None
```

Keep all original rows and also all original values

```
>>> s1.compare(s2, keep_shape=True, keep_equal=True).sort_index()
   self other
0    a      a
1    b      a
2    c      c
3    d      b
4    e      e
```

```
>>> reset_option("compute.ops_on_diff_frames")
```

## databricks.koalas.Series.replace

`Series.replace(to_replace=None, value=None, regex=False) → databricks.koalas.series.Series`

Replace values given in `to_replace` with `value`. Values of the Series are replaced with other values dynamically.

### Parameters

**to\_replace** [str, list, tuple, dict, Series, int, float, or None] How to find the values that will be replaced. \* numeric, str:

- numeric: numeric values equal to `to_replace` will be replaced with `value`
- str: string exactly matching `to_replace` will be replaced with `value`
- list of str or numeric:

- if `to_replace` and `value` are both lists or tuples, they must be the same length.
- str and numeric rules apply as above.
- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{ 'a': 'b', 'y': 'z' }` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the `value` parameter should be `None`.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{ 'a': 1, 'b': 'z' }` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in `value`. The `value` parameter should not be `None` in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.

See the examples section for examples of each of these.

**value** [scalar, dict, list, tuple, str default None] Value to replace any values matching `to_replace` with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

### Returns

**Series** Object after replacement.

## Examples

Scalar `to_replace` and `value`

```
>>> s = ks.Series([0, 1, 2, 3, 4])
>>> s
0    0
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

List-like `to_replace`

```
>>> s.replace([0, 4], 5000)
0    5000
1         1
2         2
3         3
4    5000
dtype: int64
```

```
>>> s.replace([1, 2, 3], [10, 20, 30])
0      0
1     10
2     20
3     30
4      4
dtype: int64
```

#### Dict-like *to\_replace*

```
>>> s.replace({1: 1000, 2: 2000, 3: 3000, 4: 4000})
0      0
1    1000
2    2000
3    3000
4    4000
dtype: int64
```

#### Also support for MultiIndex

```
>>> midx = pd.MultiIndex([['lama', 'cow', 'falcon'],
...                        ['speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3],
...               index=midx)
>>> s
lama      speed      45.0
          weight    200.0
          length     1.2
cow       speed      30.0
          weight    250.0
          length     1.5
falcon    speed     320.0
          weight      1.0
          length     0.3
dtype: float64
```

```
>>> s.replace(45, 450)
lama      speed      450.0
          weight    200.0
          length     1.2
cow       speed      30.0
          weight    250.0
          length     1.5
falcon    speed     320.0
          weight      1.0
          length     0.3
dtype: float64
```

```
>>> s.replace([45, 30, 320], 500)
lama      speed      500.0
          weight    200.0
          length     1.2
cow       speed      500.0
          weight    250.0
          length     1.5
```

(continues on next page)

(continued from previous page)

```
falcon  speed      500.0
        weight      1.0
        length      0.3
dtype: float64
```

```
>>> s.replace({45: 450, 30: 300})
lama    speed      450.0
        weight      200.0
        length      1.2
cow     speed      300.0
        weight      250.0
        length      1.5
falcon  speed      320.0
        weight      1.0
        length      0.3
dtype: float64
```

### databricks.koalas.Series.update

`Series.update(other) → None`

Modify Series in place using non-NA values from passed Series. Aligns on index.

#### Parameters

**other** [Series]

### Examples

```
>>> from databricks.koalas.config import set_option, reset_option
>>> set_option("compute.ops_on_diff_frames", True)
>>> s = ks.Series([1, 2, 3])
>>> s.update(ks.Series([4, 5, 6]))
>>> s.sort_index()
0    4
1    5
2    6
dtype: int64
```

```
>>> s = ks.Series(['a', 'b', 'c'])
>>> s.update(ks.Series(['d', 'e'], index=[0, 2]))
>>> s.sort_index()
0    d
1    b
2    e
dtype: object
```

```
>>> s = ks.Series([1, 2, 3])
>>> s.update(ks.Series([4, 5, 6, 7, 8]))
>>> s.sort_index()
0    4
1    5
2    6
dtype: int64
```

```
>>> s = ks.Series([1, 2, 3], index=[10, 11, 12])
>>> s
10    1
11    2
12    3
dtype: int64
```

```
>>> s.update(ks.Series([4, 5, 6]))
>>> s.sort_index()
10    1
11    2
12    3
dtype: int64
```

```
>>> s.update(ks.Series([4, 5, 6], index=[11, 12, 13]))
>>> s.sort_index()
10    1
11    4
12    5
dtype: int64
```

If other contains NaNs the corresponding values are not updated in the original Series.

```
>>> s = ks.Series([1, 2, 3])
>>> s.update(ks.Series([4, np.nan, 6]))
>>> s.sort_index()
0     4.0
1     2.0
2     6.0
dtype: float64
```

```
>>> reset_option("compute.ops_on_diff_frames")
```

### 3.3.12 Time series-related

<code>Series.asof(where)</code>	Return the last row(s) without any NaNs before <i>where</i> .
<code>Series.shift([periods, fill_value])</code>	Shift Series/Index by desired number of periods.
<code>Series.first_valid_index()</code>	Retrieves the index of the first valid value.
<code>Series.last_valid_index()</code>	Return index for last non-NA/null value.
<code>Series.at_time(time[, asof, axis])</code>	Select values at particular time of day (e.g., 9:30AM).
<code>Series.between_time(start_time, end_time[, ...])</code>	Select values between particular times of the day (e.g., 9:00-9:30 AM).

**databricks.koalas.Series.asof**

`Series.asof(when) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, databricks.koalas.series.Series]`

Return the last row(s) without any NaNs before *when*.

The last row (for each element in *when*, if list) without any NaN is taken.

If there is no good value, NaN is returned.

---

**Note:** This API is dependent on `Index.is_monotonic_increasing()` which can be expensive.

---

**Parameters**

**when** [index or array-like of indices]

**Returns**

**scalar or Series** The return can be:

- scalar : when *self* is a Series and *when* is a scalar
- Series: when *self* is a Series and *when* is an array-like

Return scalar or Series

**Notes**

Indices are assumed to be sorted. Raises if this is not the case.

**Examples**

```
>>> s = ks.Series([1, 2, np.nan, 4], index=[10, 20, 30, 40])
>>> s
10    1.0
20    2.0
30    NaN
40    4.0
dtype: float64
```

A scalar *when*.

```
>>> s.asof(20)
2.0
```

For a sequence *when*, a Series is returned. The first value is NaN, because the first element of *when* is before the first index value.

```
>>> s.asof([5, 20]).sort_index()
5      NaN
20     2.0
dtype: float64
```

Missing values are not considered. The following is 2.0, not NaN, even though NaN is at the index location for 30.

```
>>> s.asof(30)
2.0
```

## databricks.koalas.Series.shift

`Series.shift` (*periods=1*, *fill\_value=None*) → Union[databricks.koalas.series.Series, databricks.koalas.indexes.base.Index]  
Shift Series/Index by desired number of periods.

---

**Note:** the current implementation of shift uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

### Parameters

**periods** [int] Number of periods to shift. Can be positive or negative.

**fill\_value** [object, optional] The scalar value to use for newly introduced missing values. The default depends on the dtype of self. For numeric data, np.nan is used.

### Returns

Copy of input Series/Index, shifted.

## Examples

```
>>> df = ks.DataFrame({'Col1': [10, 20, 15, 30, 45],
...                     'Col2': [13, 23, 18, 33, 48],
...                     'Col3': [17, 27, 22, 37, 52]},
...                   columns=['Col1', 'Col2', 'Col3'])
```

```
>>> df.Col1.shift(periods=3)
0      NaN
1      NaN
2      NaN
3     10.0
4     20.0
Name: Col1, dtype: float64
```

```
>>> df.Col2.shift(periods=3, fill_value=0)
0      0
1      0
2      0
3     13
4     23
Name: Col2, dtype: int64
```

```
>>> df.index.shift(periods=3, fill_value=0)
Int64Index([0, 0, 0, 0, 1], dtype='int64')
```



**databricks.koalas.Series.first\_valid\_index**

`Series.first_valid_index()` → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Tuple[Union[int, float, str, bytes, decimal.Decimal, datetime.date, None], ...]]

Retrieves the index of the first valid value.

**Returns**

scalar, tuple, or None

**Examples****Support for DataFrame**

```
>>> kdf = ks.DataFrame({'a': [None, 2, 3, 2],
...                      'b': [None, 2.0, 3.0, 1.0],
...                      'c': [None, 200, 400, 200]},
...                      index=['Q', 'W', 'E', 'R'])
>>> kdf
```

	a	b	c
Q	NaN	NaN	NaN
W	2.0	2.0	200.0
E	3.0	3.0	400.0
R	2.0	1.0	200.0

```
>>> kdf.first_valid_index()
'W'
```

**Support for MultiIndex columns**

```
>>> kdf.columns = pd.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> kdf
```

	a	b	c
	x	y	z
Q	NaN	NaN	NaN
W	2.0	2.0	200.0
E	3.0	3.0	400.0
R	2.0	1.0	200.0

```
>>> kdf.first_valid_index()
'W'
```

**Support for Series.**

```
>>> s = ks.Series([None, None, 3, 4, 5], index=[100, 200, 300, 400, 500])
>>> s
```

100	NaN
200	NaN
300	3.0
400	4.0
500	5.0

dtype: float64

```
>>> s.first_valid_index()
300
```

## Support for MultiIndex

```
>>> midx = pd.MultiIndex([[ 'lama', 'cow', 'falcon'],
...                       [ 'speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> s = ks.Series([None, None, None, None, 250, 1.5, 320, 1, 0.3], index=midx)
>>> s
lama      speed      NaN
          weight      NaN
          length      NaN
cow       speed      NaN
          weight    250.0
          length     1.5
falcon    speed    320.0
          weight     1.0
          length     0.3
dtype: float64
```

```
>>> s.first_valid_index()
('cow', 'weight')
```

**databricks.koalas.Series.last\_valid\_index**

`Series.last_valid_index()` → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Tuple[Union[int, float, str, bytes, decimal.Decimal, datetime.date, None], ...]]

Return index for last non-NA/null value.

**Returns**

scalar, tuple, or None

**Notes**

This API only works with PySpark >= 3.0.

**Examples**

## Support for DataFrame

```
>>> kdf = ks.DataFrame({'a': [1, 2, 3, None],
...                     'b': [1.0, 2.0, 3.0, None],
...                     'c': [100, 200, 400, None]},
...                     index=['Q', 'W', 'E', 'R'])
>>> kdf
   a    b    c
Q  1.0  1.0 100.0
W  2.0  2.0 200.0
E  3.0  3.0 400.0
R  NaN  NaN  NaN
```

```
>>> kdf.last_valid_index()
'E'
```

## Support for MultiIndex columns

```
>>> kdf.columns = pd.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> kdf
```

	a	b	c
	x	y	z
Q	1.0	1.0	100.0
W	2.0	2.0	200.0
E	3.0	3.0	400.0
R	NaN	NaN	NaN

```
>>> kdf.last_valid_index()
'E'
```

### Support for Series.

```
>>> s = ks.Series([1, 2, 3, None, None], index=[100, 200, 300, 400, 500])
>>> s
```

100	1.0
200	2.0
300	3.0
400	NaN
500	NaN

```
dtype: float64
```

```
>>> s.last_valid_index()
300
```

### Support for MultiIndex

```
>>> midx = pd.MultiIndex([['lama', 'cow', 'falcon'],
...                        ['speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> s = ks.Series([250, 1.5, 320, 1, 0.3, None, None, None, None], index=midx)
>>> s
```

lama	speed	250.0
	weight	1.5
	length	320.0
cow	speed	1.0
	weight	0.3
	length	NaN
falcon	speed	NaN
	weight	NaN
	length	NaN

```
dtype: float64
```

```
>>> s.last_valid_index()
('cow', 'weight')
```

**databricks.koalas.Series.at\_time**

`Series.at_time` (*time*: Union[datetime.time, str], *asof*: bool = False, *axis*: Union[int, str] = 0) → databricks.koalas.series.Series  
 Select values at particular time of day (e.g., 9:30AM).

**Parameters**

**time** [datetime.time or str]  
**axis** [{0 or 'index', 1 or 'columns'}, default 0]

**Returns**

**Series**

**Raises**

**TypeError** If the index is not a *DatetimeIndex*

See also:

*between\_time* Select values between particular times of the day.

*DatetimeIndex.indexer\_at\_time* Get just the index locations for values at particular time of the day.

**Examples**

```
>>> idx = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> kser = ks.Series([1, 2, 3, 4], index=idx)
>>> kser
2018-04-09 00:00:00    1
2018-04-09 12:00:00    2
2018-04-10 00:00:00    3
2018-04-10 12:00:00    4
dtype: int64
```

```
>>> kser.at_time('12:00')
2018-04-09 12:00:00    2
2018-04-10 12:00:00    4
dtype: int64
```

**databricks.koalas.Series.between\_time**

`Series.between_time` (*start\_time*: Union[datetime.time, str], *end\_time*: Union[datetime.time, str], *include\_start*: bool = True, *include\_end*: bool = True, *axis*: Union[int, str] = 0) → databricks.koalas.series.Series

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting *start\_time* to be later than *end\_time*, you can get the times that are *not* between the two times.

**Parameters**

**start\_time** [datetime.time or str] Initial time as a time filter limit.  
**end\_time** [datetime.time or str] End time as a time filter limit.  
**include\_start** [bool, default True] Whether the start time needs to be included in the result.  
**include\_end** [bool, default True] Whether the end time needs to be included in the result.

**axis** [{0 or 'index', 1 or 'columns'}, default 0] Determine range time on index or columns value.

### Returns

**Series** Data from the original object filtered to the specified dates range.

### Raises

**TypeError** If the index is not a *DatetimeIndex*

### See also:

*at\_time* Select values at a particular time of the day.

*last* Select final periods of time series based on a date offset.

*DatetimeIndex.indexer\_between\_time* Get just the index locations for values between particular times of the day.

### Examples

```
>>> idx = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> kser = ks.Series([1, 2, 3, 4], index=idx)
>>> kser
2018-04-09 00:00:00    1
2018-04-10 00:20:00    2
2018-04-11 00:40:00    3
2018-04-12 01:00:00    4
dtype: int64
```

```
>>> kser.between_time('0:15', '0:45')
2018-04-10 00:20:00    2
2018-04-11 00:40:00    3
dtype: int64
```

## 3.3.13 Spark-related

`Series.spark` provides features that does not exist in pandas but in Spark. These can be accessed by `Series.spark.<function/property>`.

<i>Series.spark.data_type</i>	Returns the data type as defined by Spark, as a Spark <code>DataType</code> object.
<i>Series.spark.nullable</i>	Returns the nullability as defined by Spark.
<i>Series.spark.column</i>	Spark Column object representing the Series/Index.
<i>Series.spark.transform(func)</i>	Applies a function that takes and returns a Spark column.
<i>Series.spark.apply(func)</i>	Applies a function that takes and returns a Spark column.

**databricks.koalas.Series.spark.data\_type****property** `spark.data_type`

Returns the data type as defined by Spark, as a Spark DataType object.

**databricks.koalas.Series.spark.nullable****property** `spark.nullable`

Returns the nullability as defined by Spark.

**databricks.koalas.Series.spark.column****property** `spark.column`

Spark Column object representing the Series/Index.

---

**Note:** This Spark Column object is strictly stick to its base DataFrame the Series/Index was derived from.

---

**databricks.koalas.Series.spark.transform**`spark.transform(func) → ks.Series`

Applies a function that takes and returns a Spark column. It allows to natively apply a Spark function and column APIs with the Spark column internally used in Series or Index. The output length of the Spark column should be same as input's.

---

**Note:** It requires to have the same input and output length; therefore, the aggregate Spark functions such as count does not work.

---

**Parameters****func** [function] Function to use for transforming the data by using Spark columns.**Returns****Series or Index****Raises****ValueError** [If the output from the function is not a Spark column.]**Examples**

```
>>> from pyspark.sql.functions import log
>>> df = ks.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]}, columns=["a", "b"])
>>> df
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.a.spark.transform(lambda c: log(c))
0    0.000000
1    0.693147
2    1.098612
Name: a, dtype: float64
```

```
>>> df.index.spark.transform(lambda c: c + 10)
Int64Index([10, 11, 12], dtype='int64')
```

```
>>> df.a.spark.transform(lambda c: c + df.b.spark.column)
0    5
1    7
2    9
Name: a, dtype: int64
```

### databricks.koalas.Series.spark.apply

`spark.apply(func) → ks.Series`

Applies a function that takes and returns a Spark column. It allows to natively apply a Spark function and column APIs with the Spark column internally used in Series or Index.

---

**Note:** It forces to lose the index and end up with using default index. It is preferred to use `Series.spark.transform()` or `meth:DataFrame.spark.apply` with specifying the `index_col`.

---



---

**Note:** It does not require to have the same length of the input and output. However, it requires to create a new DataFrame internally which will require to set `compute.ops_on_diff_frames` to compute even with the same origin DataFrame that is expensive, whereas `Series.spark.transform()` does not require it.

---

#### Parameters

**func** [function] Function to apply the function against the data by using Spark columns.

#### Returns

**Series**

#### Raises

**ValueError** [If the output from the function is not a Spark column.]

### Examples

```
>>> from databricks import koalas as ks
>>> from pyspark.sql.functions import count, lit
>>> df = ks.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]}, columns=["a", "b"])
>>> df
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.a.spark.apply(lambda c: count(c))
0      3
Name: a, dtype: int64
```

```
>>> df.a.spark.apply(lambda c: c + df.b.spark.column)
0      5
1      7
2      9
Name: a, dtype: int64
```

### 3.3.14 Accessors

Koalas provides dtype-specific methods under various accessors. These are separate namespaces within *Series* that only apply to specific data types.

Data Type	Accessor
Datetime	<i>dt</i>
String	<i>str</i>
Categorical	<i>cat</i>

### 3.3.15 Date Time Handling

*Series.dt* can be used to access the values of the series as datetimelike and return several properties. These can be accessed like *Series.dt.<property>*.

#### Datetime Properties

<i>Series.dt.date</i>	Returns a Series of python <code>datetime.date</code> objects (namely, the date part of Timestamps without timezone information).
<i>Series.dt.year</i>	The year of the datetime.
<i>Series.dt.month</i>	The month of the timestamp as January = 1 December = 12.
<i>Series.dt.day</i>	The days of the datetime.
<i>Series.dt.hour</i>	The hours of the datetime.
<i>Series.dt.minute</i>	The minutes of the datetime.
<i>Series.dt.second</i>	The seconds of the datetime.
<i>Series.dt.microsecond</i>	The microseconds of the datetime.
<i>Series.dt.week</i>	The week ordinal of the year.
<i>Series.dt.weekofyear</i>	The week ordinal of the year.
<i>Series.dt.dayofweek</i>	The day of the week with Monday=0, Sunday=6.
<i>Series.dt.weekday</i>	The day of the week with Monday=0, Sunday=6.
<i>Series.dt.dayofyear</i>	The ordinal day of the year.
<i>Series.dt.quarter</i>	The quarter of the date.
<i>Series.dt.is_month_start</i>	Indicates whether the date is the first day of the month.
<i>Series.dt.is_month_end</i>	Indicates whether the date is the last day of the month.
<i>Series.dt.is_quarter_start</i>	Indicator for whether the date is the first day of a quarter.
<i>Series.dt.is_quarter_end</i>	Indicator for whether the date is the last day of a quarter.

continues on next page



Table 32 – continued from previous page

<i>Series.dt.is_year_start</i>	Indicate whether the date is the first day of a year.
<i>Series.dt.is_year_end</i>	Indicate whether the date is the last day of the year.
<i>Series.dt.is_leap_year</i>	Boolean indicator if the date belongs to a leap year.
<i>Series.dt.daysinmonth</i>	The number of days in the month.
<i>Series.dt.days_in_month</i>	The number of days in the month.

**databricks.koalas.Series.dt.date****property** `dt.date`

Returns a Series of python datetime.date objects (namely, the date part of Timestamps without timezone information).

**databricks.koalas.Series.dt.year****property** `dt.year`

The year of the datetime.

**databricks.koalas.Series.dt.month****property** `dt.month`

The month of the timestamp as January = 1 December = 12.

**databricks.koalas.Series.dt.day****property** `dt.day`

The days of the datetime.

**databricks.koalas.Series.dt.hour****property** `dt.hour`

The hours of the datetime.

**databricks.koalas.Series.dt.minute****property** `dt.minute`

The minutes of the datetime.

**databricks.koalas.Series.dt.second****property** `dt.second`

The seconds of the datetime.

**databricks.koalas.Series.dt.microsecond****property** `dt.microsecond`

The microseconds of the datetime.

**databricks.koalas.Series.dt.week****property** `dt.week`

The week ordinal of the year.

**databricks.koalas.Series.dt.weekofyear****property** `dt.weekofyear`

The week ordinal of the year.

**databricks.koalas.Series.dt.dayofweek****property** `dt.dayofweek`

The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor).

**Returns****Series** Containing integers indicating the day number.**See also:****Series.dt.dayofweek** Alias.**Series.dt.weekday** Alias.**Series.dt.day\_name** Returns the name of the day of the week.**Examples**

```
>>> s = ks.from_pandas(pd.date_range('2016-12-31', '2017-01-08', freq='D').to_
↳series())
>>> s.dt.dayofweek
2016-12-31    5
2017-01-01    6
2017-01-02    0
2017-01-03    1
2017-01-04    2
2017-01-05    3
2017-01-06    4
```

(continues on next page)

(continued from previous page)

```

2017-01-07    5
2017-01-08    6
dtype: int64

```

### **databricks.koalas.Series.dt.weekday**

#### **property** `dt.weekday`

The day of the week with Monday=0, Sunday=6.

Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor).

#### **Returns**

**Series** Containing integers indicating the day number.

**See also:**

**Series.dt.dayofweek** Alias.

**Series.dt.weekday** Alias.

**Series.dt.day\_name** Returns the name of the day of the week.

### **Examples**

```

>>> s = ks.from_pandas(pd.date_range('2016-12-31', '2017-01-08', freq='D').to_
↳series())
>>> s.dt.dayofweek
2016-12-31    5
2017-01-01    6
2017-01-02    0
2017-01-03    1
2017-01-04    2
2017-01-05    3
2017-01-06    4
2017-01-07    5
2017-01-08    6
dtype: int64

```

### **databricks.koalas.Series.dt.dayofyear**

#### **property** `dt.dayofyear`

The ordinal day of the year.

**databricks.koalas.Series.dt.quarter****property** `dt.quarter`

The quarter of the date.

**databricks.koalas.Series.dt.is\_month\_start****property** `dt.is_month_start`

Indicates whether the date is the first day of the month.

**Returns****Series** For Series, returns a Series with boolean values.**See also:****`is_month_end`** Return a boolean indicating whether the date is the last day of the month.**Examples**This method is available on Series with datetime values under the `.dt` accessor.

```
>>> s = ks.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0    2018-02-27
1    2018-02-28
2    2018-03-01
dtype: datetime64[ns]
```

```
>>> s.dt.is_month_start
0    False
1    False
2     True
dtype: bool
```

**databricks.koalas.Series.dt.is\_month\_end****property** `dt.is_month_end`

Indicates whether the date is the last day of the month.

**Returns****Series** For Series, returns a Series with boolean values.**See also:****`is_month_start`** Return a boolean indicating whether the date is the first day of the month.

## Examples

This method is available on Series with datetime values under the `.dt` accessor.

```
>>> s = ks.Series(pd.date_range("2018-02-27", periods=3))
>>> s
0    2018-02-27
1    2018-02-28
2    2018-03-01
dtype: datetime64[ns]
```

```
>>> s.dt.is_month_end
0    False
1     True
2    False
dtype: bool
```

## `databricks.koalas.Series.dt.is_quarter_start`

### property `dt.is_quarter_start`

Indicator for whether the date is the first day of a quarter.

#### Returns

**`is_quarter_start`** [Series] The same type as the original data with boolean values. Series will have the same name and index.

See also:

[`quarter`](#) Return the quarter of the date.

[`is\_quarter\_end`](#) Similar property for indicating the quarter start.

## Examples

This method is available on Series with datetime values under the `.dt` accessor.

```
>>> df = ks.DataFrame({'dates': pd.date_range("2017-03-30",
...                                           periods=4)})
>>> df
   dates
0 2017-03-30
1 2017-03-31
2 2017-04-01
3 2017-04-02
```

```
>>> df.dates.dt.quarter
0    1
1    1
2    2
3    2
Name: dates, dtype: int64
```

```
>>> df.dates.dt.is_quarter_start
0    False
1    False
2     True
3    False
Name: dates, dtype: bool
```

### `databricks.koalas.Series.dt.is_quarter_end`

#### **property** `dt.is_quarter_end`

Indicator for whether the date is the last day of a quarter.

#### **Returns**

**is\_quarter\_end** [Series] The same type as the original data with boolean values. Series will have the same name and index.

#### **See also:**

[`quarter`](#) Return the quarter of the date.

[`is\_quarter\_start`](#) Similar property indicating the quarter start.

### **Examples**

This method is available on Series with datetime values under the `.dt` accessor.

```
>>> df = ks.DataFrame({'dates': pd.date_range("2017-03-30",
...                                           periods=4)})
>>> df
   dates
0 2017-03-30
1 2017-03-31
2 2017-04-01
3 2017-04-02
```

```
>>> df.dates.dt.quarter
0    1
1    1
2    2
3    2
Name: dates, dtype: int64
```

```
>>> df.dates.dt.is_quarter_start
0    False
1    False
2     True
3    False
Name: dates, dtype: bool
```

**databricks.koalas.Series.dt.is\_year\_start****property** `dt.is_year_start`

Indicate whether the date is the first day of a year.

**Returns**

**Series** The same type as the original data with boolean values. Series will have the same name and index.

**See also:**

[\*is\\_year\\_end\*](#) Similar property indicating the last day of the year.

**Examples**

This method is available on Series with datetime values under the `.dt` accessor.

```
>>> dates = ks.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0    2017-12-30
1    2017-12-31
2    2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_start
0    False
1    False
2     True
dtype: bool
```

**databricks.koalas.Series.dt.is\_year\_end****property** `dt.is_year_end`

Indicate whether the date is the last day of the year.

**Returns**

**Series** The same type as the original data with boolean values. Series will have the same name and index.

**See also:**

[\*is\\_year\\_start\*](#) Similar property indicating the start of the year.

## Examples

This method is available on Series with datetime values under the `.dt` accessor.

```
>>> dates = ks.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0    2017-12-30
1    2017-12-31
2    2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_end
0    False
1     True
2    False
dtype: bool
```

## `databricks.koalas.Series.dt.is_leap_year`

### property `dt.is_leap_year`

Boolean indicator if the date belongs to a leap year.

A leap year is a year, which has 366 days (instead of 365) including 29th of February as an intercalary day. Leap years are years which are multiples of four with the exception of years divisible by 100 but not by 400.

#### Returns

**Series** Booleans indicating if dates belong to a leap year.

## Examples

This method is available on Series with datetime values under the `.dt` accessor.

```
>>> dates_series = ks.Series(pd.date_range("2012-01-01", "2015-01-01", freq="Y"))
>>> dates_series
0    2012-12-31
1    2013-12-31
2    2014-12-31
dtype: datetime64[ns]
```

```
>>> dates_series.dt.is_leap_year
0     True
1    False
2    False
dtype: bool
```



**databricks.koalas.Series.dt.daysinmonth****property** `dt.daysinmonth`

The number of days in the month.

**databricks.koalas.Series.dt.days\_in\_month****property** `dt.days_in_month`

The number of days in the month.

**Datetime Methods**

<code>Series.dt.normalize()</code>	Convert times to midnight.
<code>Series.dt.strftime(date_format)</code>	Convert to a string Series using specified date_format.
<code>Series.dt.round(freq, *args, **kwargs)</code>	Perform round operation on the data to the specified freq.
<code>Series.dt.floor(freq, *args, **kwargs)</code>	Perform floor operation on the data to the specified freq.
<code>Series.dt.ceil(freq, *args, **kwargs)</code>	Perform ceil operation on the data to the specified freq.
<code>Series.dt.month_name([locale])</code>	Return the month names of the series with specified locale.
<code>Series.dt.day_name([locale])</code>	Return the day names of the series with specified locale.

**databricks.koalas.Series.dt.normalize**`dt.normalize()` → `ks.Series`

Convert times to midnight.

The time component of the date-time is converted to midnight i.e. 00:00:00. This is useful in cases, when the time does not matter. Length is unaltered. The timezones are unaffected.

This method is available on Series with datetime values under the `.dt` accessor, and directly on Datetime Array.

**Returns**

**Series** The same type as the original data. Series will have the same name and index.

**See also:**

**floor** Floor the series to the specified freq.

**ceil** Ceil the series to the specified freq.

**round** Round the series to the specified freq.

## Examples

```
>>> series = ks.Series(pd.Series(pd.date_range('2012-1-1 12:45:31', periods=3,
↪freq='M'))))
>>> series.dt.normalize()
0    2012-01-31
1    2012-02-29
2    2012-03-31
dtype: datetime64[ns]
```

## databricks.koalas.Series.dt.strftime

`dt.strftime(date_format) → ks.Series`

Convert to a string Series using specified date\_format.

Return an series of formatted strings specified by date\_format, which supports the same string format as the python standard library. Details of the string format can be found in python string format doc.

### Parameters

**date\_format** [str] Date format string (e.g. “%%Y-%%m-%%d”).

### Returns

**Series** Series of formatted strings.

See also:

**to\_datetime** Convert the given argument to datetime.

**normalize** Return series with times to midnight.

**round** Round the series to the specified freq.

**floor** Floor the series to the specified freq.

## Examples

```
>>> series = ks.Series(pd.date_range(pd.Timestamp("2018-03-10 09:00"),
...                                   periods=3, freq='s'))
>>> series
0    2018-03-10 09:00:00
1    2018-03-10 09:00:01
2    2018-03-10 09:00:02
dtype: datetime64[ns]
```

```
>>> series.dt.strftime('%B %d, %Y, %r')
0    March 10, 2018, 09:00:00 AM
1    March 10, 2018, 09:00:01 AM
2    March 10, 2018, 09:00:02 AM
dtype: object
```

**databricks.koalas.Series.dt.round**

`dt.round(freq, *args, **kwargs) → ks.Series`

Perform round operation on the data to the specified freq.

**Parameters**

**freq** [str or Offset] The frequency level to round the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end).

**nonexistent** ['shift\_forward', 'shift\_backward', 'NaT', timedelta, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift\_forward' will shift the nonexistent time forward to the closest existing time
- 'shift\_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an `NonExistentTimeError` if there are nonexistent times

---

**Note:** this option only works with pandas 0.24.0+

---

**Returns**

**Series** a Series with the same index for a Series.

**Raises**

**ValueError** if the *freq* cannot be converted.

**Examples**

```
>>> series = ks.Series(pd.date_range('1/1/2018 11:59:00', periods=3, freq='min'))
>>> series
0    2018-01-01 11:59:00
1    2018-01-01 12:00:00
2    2018-01-01 12:01:00
dtype: datetime64[ns]
```

```
>>> series.dt.round("H")
0    2018-01-01 12:00:00
1    2018-01-01 12:00:00
2    2018-01-01 12:00:00
dtype: datetime64[ns]
```

**databricks.koalas.Series.dt.floor**

`dt.floor(freq, *args, **kwargs) → ks.Series`

Perform floor operation on the data to the specified freq.

**Parameters**

**freq** [str or Offset] The frequency level to floor the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end).

**nonexistent** ['shift\_forward', 'shift\_backward', 'NaT', timedelta, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift\_forward' will shift the nonexistent time forward to the closest existing time
- 'shift\_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an `NonExistentTimeError` if there are nonexistent times

---

**Note:** this option only works with pandas 0.24.0+

---

**Returns**

**Series** a Series with the same index for a Series.

**Raises**

**ValueError** if the *freq* cannot be converted.

**Examples**

```
>>> series = ks.Series(pd.date_range('1/1/2018 11:59:00', periods=3, freq='min'))
>>> series
0    2018-01-01 11:59:00
1    2018-01-01 12:00:00
2    2018-01-01 12:01:00
dtype: datetime64[ns]
```

```
>>> series.dt.floor("H")
0    2018-01-01 11:00:00
1    2018-01-01 12:00:00
2    2018-01-01 12:00:00
dtype: datetime64[ns]
```

**databricks.koalas.Series.dt.ceil**

`dt.ceil(freq, *args, **kwargs) → ks.Series`

Perform ceil operation on the data to the specified freq.

**Parameters**

**freq** [str or Offset] The frequency level to round the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end).

**nonexistent** ['shift\_forward', 'shift\_backward', 'NaT', timedelta, default 'raise'] A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST.

- 'shift\_forward' will shift the nonexistent time forward to the closest existing time
- 'shift\_backward' will shift the nonexistent time backward to the closest existing time
- 'NaT' will return NaT where there are nonexistent times
- timedelta objects will shift nonexistent times by the timedelta
- 'raise' will raise an `NonExistentTimeError` if there are nonexistent times

---

**Note:** this option only works with pandas 0.24.0+

---

**Returns**

**Series** a Series with the same index for a Series.

**Raises**

**ValueError** if the *freq* cannot be converted.

**Examples**

```
>>> series = ks.Series(pd.date_range('1/1/2018 11:59:00', periods=3, freq='min'))
>>> series
0    2018-01-01 11:59:00
1    2018-01-01 12:00:00
2    2018-01-01 12:01:00
dtype: datetime64[ns]
```

```
>>> series.dt.ceil("H")
0    2018-01-01 12:00:00
1    2018-01-01 12:00:00
2    2018-01-01 13:00:00
dtype: datetime64[ns]
```

**databricks.koalas.Series.dt.month\_name**

`dt.month_name(locale=None) → ks.Series`

Return the month names of the series with specified locale.

**Parameters**

**locale** [str, optional] Locale determining the language in which to return the month name. Default is English locale.

**Returns**

**Series** Series of month names.

**Examples**

```
>>> series = ks.Series(pd.date_range(start='2018-01', freq='M', periods=3))
>>> series
0    2018-01-31
1    2018-02-28
2    2018-03-31
dtype: datetime64[ns]
```

```
>>> series.dt.month_name()
0    January
1    February
2    March
dtype: object
```

**databricks.koalas.Series.dt.day\_name**

`dt.day_name(locale=None) → ks.Series`

Return the day names of the series with specified locale.

**Parameters**

**locale** [str, optional] Locale determining the language in which to return the day name. Default is English locale.

**Returns**

**Series** Series of day names.

**Examples**

```
>>> series = ks.Series(pd.date_range(start='2018-01-01', freq='D', periods=3))
>>> series
0    2018-01-01
1    2018-01-02
2    2018-01-03
dtype: datetime64[ns]
```

```
>>> series.dt.day_name()
0      Monday
1     Tuesday
2    Wednesday
dtype: object
```

### 3.3.16 String Handling

`Series.str` can be used to access the values of the series as strings and apply several methods to it. These can be accessed like `Series.str.<function/property>`.

<code>Series.str.capitalize()</code>	Convert Strings in the series to be capitalized.
<code>Series.str.cat([others, sep, na_rep, join])</code>	Not supported.
<code>Series.str.center(width[, fillchar])</code>	Filling left and right side of strings in the Series/Index with an additional character.
<code>Series.str.contains(pat[, case, flags, na, ...])</code>	Test if pattern or regex is contained within a string of a Series.
<code>Series.str.count(pat[, flags])</code>	Count occurrences of pattern in each string of the Series.
<code>Series.str.decode(encoding[, errors])</code>	Not supported.
<code>Series.str.encode(encoding[, errors])</code>	Not supported.
<code>Series.str.endswith(pattern[, na])</code>	Test if the end of each string element matches a pattern.
<code>Series.str.extract(pat[, flags, expand])</code>	Not supported.
<code>Series.str.extractall(pat[, flags])</code>	Not supported.
<code>Series.str.find(sub[, start, end])</code>	Return lowest indexes in each strings in the Series where the substring is fully contained between [start:end].
<code>Series.str.findall(pat[, flags])</code>	Find all occurrences of pattern or regular expression in the Series.
<code>Series.str.get(i)</code>	Extract element from each string or string list/tuple in the Series at the specified position.
<code>Series.str.get_dummies([sep])</code>	Not supported.
<code>Series.str.index(sub[, start, end])</code>	Return lowest indexes in each strings where the substring is fully contained between [start:end].
<code>Series.str.isalnum()</code>	Check whether all characters in each string are alphanumeric.
<code>Series.str.isalpha()</code>	Check whether all characters in each string are alphabetic.
<code>Series.str.isdigit()</code>	Check whether all characters in each string are digits.
<code>Series.str.isspace()</code>	Check whether all characters in each string are whitespaces.
<code>Series.str.islower()</code>	Check whether all characters in each string are lowercase.
<code>Series.str.isupper()</code>	Check whether all characters in each string are uppercase.
<code>Series.str.istitle()</code>	Check whether all characters in each string are titlecase.
<code>Series.str.isnumeric()</code>	Check whether all characters in each string are numeric.
<code>Series.str.isdecimal()</code>	Check whether all characters in each string are decimals.
<code>Series.str.join(sep)</code>	Join lists contained as elements in the Series with passed delimiter.
<code>Series.str.len()</code>	Computes the length of each element in the Series.

continues on next page

Table 34 – continued from previous page

<code>Series.str.ljust(width[, fillchar])</code>	Filling right side of strings in the Series with an additional character.
<code>Series.str.lower()</code>	Convert strings in the Series/Index to all lowercase.
<code>Series.str.lstrip([to_strip])</code>	Remove leading characters.
<code>Series.str.match(pat[, case, flags, na])</code>	Determine if each string matches a regular expression.
<code>Series.str.normalize(form)</code>	Return the Unicode normal form for the strings in the Series.
<code>Series.str.pad(width[, side, fillchar])</code>	Pad strings in the Series up to width.
<code>Series.str.partition([sep, expand])</code>	Not supported.
<code>Series.str.repeat(repeats)</code>	Duplicate each string in the Series.
<code>Series.str.replace(pat, repl[, n, case, ...])</code>	Replace occurrences of pattern/regex in the Series with some other string.
<code>Series.str.rfind(sub[, start, end])</code>	Return highest indexes in each strings in the Series where the substring is fully contained between [start:end].
<code>Series.str.rindex(sub[, start, end])</code>	Return highest indexes in each strings where the substring is fully contained between [start:end].
<code>Series.str.rjust(width[, fillchar])</code>	Filling left side of strings in the Series with an additional character.
<code>Series.str.rpartition([sep, expand])</code>	Not supported.
<code>Series.str.rsplit([pat, n, expand])</code>	Split strings around given separator/delimiter.
<code>Series.str.rstrip([to_strip])</code>	Remove trailing characters.
<code>Series.str.slice([start, stop, step])</code>	Slice substrings from each element in the Series.
<code>Series.str.slice_replace([start, stop, repl])</code>	Slice substrings from each element in the Series.
<code>Series.str.split([pat, n, expand])</code>	Split strings around given separator/delimiter.
<code>Series.str.startswith(pattern[, na])</code>	Test if the start of each string element matches a pattern.
<code>Series.str.strip([to_strip])</code>	Remove leading and trailing characters.
<code>Series.str.swapcase()</code>	Convert strings in the Series/Index to be swapcased.
<code>Series.str.title()</code>	Convert Strings in the series to be titlecase.
<code>Series.str.translate(table)</code>	Map all characters in the string through the given mapping table.
<code>Series.str.upper()</code>	Convert strings in the Series/Index to all uppercase.
<code>Series.str.wrap(width, **kwargs)</code>	Wrap long strings in the Series to be formatted in paragraphs with length less than a given width.
<code>Series.str.zfill(width)</code>	Pad strings in the Series by prepending '0' characters.

**databricks.koalas.Series.str.capitalize**

`str.capitalize()` → `ks.Series`

Convert Strings in the series to be capitalized.



## Examples

```
>>> s = ks.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0          lower
1        CAPITALS
2    this is a sentence
3        SwApCaSe
dtype: object
```

```
>>> s.str.capitalize()
0          Lower
1        Capitals
2    This is a sentence
3        Swapcase
dtype: object
```

### `databricks.koalas.Series.str.cat`

`str.cat` (*others=None, sep=None, na\_rep=None, join=None*) → `ks.Series`  
Not supported.

### `databricks.koalas.Series.str.center`

`str.center` (*width, fillchar=' '*) → `ks.Series`  
Filling left and right side of strings in the Series/Index with an additional character. Equivalent to `str.center()`.

#### Parameters

**width** [int] Minimum width of resulting string; additional characters will be filled with `fillchar`.

**fillchar** [str] Additional character for filling, default is whitespace.

#### Returns

Series of objects

## Examples

```
>>> s = ks.Series(["caribou", "tiger"])
>>> s
0    caribou
1     tiger
dtype: object
```

```
>>> s.str.center(width=10, fillchar='-')
0    -caribou--
1    --tiger---
dtype: object
```

**databricks.koalas.Series.str.contains**

`str.contains(pat, case=True, flags=0, na=None, regex=True) → ks.Series`

Test if pattern or regex is contained within a string of a Series.

Return boolean Series based on whether a given pattern or regex is contained within a string of a Series.

Analogous to `match()`, but less strict, relying on `re.search()` instead of `re.match()`.

**Parameters**

**pat** [str] Character sequence or regular expression.

**case** [bool, default True] If True, case sensitive.

**flags** [int, default 0 (no flags)] Flags to pass through to the re module, e.g. `re.IGNORECASE`.

**na** [default None] Fill value for missing values. NaN converted to None.

**regex** [bool, default True] If True, assumes the pat is a regular expression. If False, treats the pat as a literal string.

**Returns**

**Series of boolean values or object** A Series of boolean values indicating whether the given pattern is contained within the string of each element of the Series.

**Examples**

Returning a Series of booleans using only a literal pattern.

```
>>> s1 = ks.Series(['Mouse', 'dog', 'house and parrot', '23', np.NaN])
>>> s1.str.contains('og', regex=False)
0    False
1     True
2    False
3    False
4     None
dtype: object
```

Specifying case sensitivity using case.

```
>>> s1.str.contains('oG', case=True, regex=True)
0    False
1    False
2    False
3    False
4     None
dtype: object
```

Specifying na to be False instead of NaN replaces NaN values with False. If Series does not contain NaN values the resultant dtype will be bool, otherwise, an object dtype.

```
>>> s1.str.contains('og', na=False, regex=True)
0    False
1     True
2    False
3    False
4    False
dtype: bool
```

Returning 'house' or 'dog' when either expression occurs in a string.

```
>>> s1.str.contains('house|dog', regex=True)
0    False
1     True
2     True
3    False
4     None
dtype: object
```

Ignoring case sensitivity using flags with regex.

```
>>> import re
>>> s1.str.contains('PARROT', flags=re.IGNORECASE, regex=True)
0    False
1    False
2     True
3    False
4     None
dtype: object
```

Returning any digit using regular expression.

```
>>> s1.str.contains('[0-9]', regex=True)
0    False
1    False
2    False
3     True
4     None
dtype: object
```

Ensure pat is not a literal pattern when regex is set to True. Note in the following example one might expect only s2[1] and s2[3] to return True. However, '.' as a regex matches any character followed by a 0.

```
>>> s2 = ks.Series(['40', '40.0', '41', '41.0', '35'])
>>> s2.str.contains('.0', regex=True)
0     True
1     True
2    False
3     True
4    False
dtype: bool
```

### **databricks.koalas.Series.str.count**

`str.count(pat, flags=0) → ks.Series`

Count occurrences of pattern in each string of the Series.

This function is used to count the number of times a particular regex pattern is repeated in each of the string elements of the Series.

#### **Parameters**

**pat** [str] Valid regular expression.

**flags** [int, default 0 (no flags)] Flags for the re module.

#### **Returns**

**Series of int** A Series containing the integer counts of pattern matches.

### Examples

```
>>> s = ks.Series(['A', 'B', 'Aaba', 'Baca', np.NaN, 'CABA', 'cat'])
>>> s.str.count('a')
0    0.0
1    0.0
2    2.0
3    2.0
4    NaN
5    0.0
6    1.0
dtype: float64
```

Escape '\$' to find the literal dollar sign.

```
>>> s = ks.Series(['$', 'B', 'Aab$', '$$ca', 'C$B$', 'cat'])
>>> s.str.count('\$')
0    1
1    0
2    1
3    2
4    2
5    0
dtype: int64
```

### `databricks.koalas.Series.str.decode`

`str.decode(encoding, errors='strict') → ks.Series`  
Not supported.

### `databricks.koalas.Series.str.encode`

`str.encode(encoding, errors='strict') → ks.Series`  
Not supported.

### `databricks.koalas.Series.str.endswith`

`str.endswith(pattern, na=None) → ks.Series`  
Test if the end of each string element matches a pattern.  
Equivalent to `str.endswith()`.

#### Parameters

**pattern** [str] Character sequence. Regular expressions are not accepted.

**na** [object, default None] Object shown if element is not a string. NaN converted to None.

#### Returns

**Series of bool or object** Koalas Series of booleans indicating whether the given pattern matches the end of each string element.

## Examples

```
>>> s = ks.Series(['bat', 'Bear', 'cat', np.nan])
>>> s
0      bat
1     Bear
2      cat
3     None
dtype: object
```

```
>>> s.str.endswith('t')
0      True
1     False
2      True
3     None
dtype: object
```

Specifying na to be False instead of None.

```
>>> s.str.endswith('t', na=False)
0      True
1     False
2      True
3     False
dtype: bool
```

### **databricks.koalas.Series.str.extract**

`str.extract(pat, flags=0, expand=True) → ks.Series`  
Not supported.

### **databricks.koalas.Series.str.extractall**

`str.extractall(pat, flags=0) → ks.Series`  
Not supported.

### **databricks.koalas.Series.str.find**

`str.find(sub, start=0, end=None) → ks.Series`  
Return lowest indexes in each strings in the Series where the substring is fully contained between [start:end].  
Return -1 on failure. Equivalent to standard `str.find()`.

#### **Parameters**

- sub** [str] Substring being searched.
- start** [int] Left edge index.
- end** [int] Right edge index.

#### **Returns**

**Series of int** Series of lowest matching indexes.

## Examples

```
>>> s = ks.Series(['apple', 'oranges', 'bananas'])
```

```
>>> s.str.find('a')
0    0
1    2
2    1
dtype: int64
```

```
>>> s.str.find('a', start=2)
0   -1
1    2
2    3
dtype: int64
```

```
>>> s.str.find('a', end=1)
0    0
1   -1
2   -1
dtype: int64
```

```
>>> s.str.find('a', start=2, end=2)
0   -1
1   -1
2   -1
dtype: int64
```

### **databricks.koalas.Series.str.findall**

`str.findall (pat, flags=0) → ks.Series`

Find all occurrences of pattern or regular expression in the Series.

Equivalent to applying `re.findall()` to all the elements in the Series.

#### **Parameters**

**pat** [str] Pattern or regular expression.

**flags** [int, default 0 (no flags)] *re* module flags, e.g. *re.IGNORECASE*.

#### **Returns**

**Series of object** All non-overlapping matches of pattern or regular expression in each string of this Series.

## Examples

```
>>> s = ks.Series(['Lion', 'Monkey', 'Rabbit'])
```

The search for the pattern 'Monkey' returns one match:

```
>>> s.str.findall('Monkey')
0      []
1    [Monkey]
2      []
dtype: object
```

On the other hand, the search for the pattern 'MONKEY' doesn't return any match:

```
>>> s.str.findall('MONKEY')
0      []
1      []
2      []
dtype: object
```

Flags can be added to the pattern or regular expression. For instance, to find the pattern 'MONKEY' ignoring the case:

```
>>> import re
>>> s.str.findall('MONKEY', flags=re.IGNORECASE)
0      []
1    [Monkey]
2      []
dtype: object
```

When the pattern matches more than one string in the Series, all matches are returned:

```
>>> s.str.findall('on')
0    [on]
1    [on]
2     []
dtype: object
```

Regular expressions are supported too. For instance, the search for all the strings ending with the word 'on' is shown next:

```
>>> s.str.findall('on$')
0    [on]
1     []
2     []
dtype: object
```

If the pattern is found more than once in the same string, then a list of multiple strings is returned:

```
>>> s.str.findall('b')
0      []
1      []
2    [b, b]
dtype: object
```

**databricks.koalas.Series.str.get**

`str.get(i) → ks.Series`

Extract element from each string or string list/tuple in the Series at the specified position.

**Parameters**

**i** [int] Position of element to extract.

**Returns**

**Series of objects**

**Examples**

```
>>> s1 = ks.Series(["String", "123"])
>>> s1
0    String
1      123
dtype: object
```

```
>>> s1.str.get(1)
0    t
1    2
dtype: object
```

```
>>> s1.str.get(-1)
0    g
1    3
dtype: object
```

```
>>> s2 = ks.Series([["a", "b", "c"], ["x", "y"]])
>>> s2
0    [a, b, c]
1    [x, y]
dtype: object
```

```
>>> s2.str.get(0)
0    a
1    x
dtype: object
```

```
>>> s2.str.get(2)
0    c
1    None
dtype: object
```



**databricks.koalas.Series.str.get\_dummies**

`str.get_dummies(sep='')`  
Not supported.

**databricks.koalas.Series.str.index**

`str.index(sub, start=0, end=None) → ks.Series`

Return lowest indexes in each strings where the substring is fully contained between [start:end].

This is the same as `str.find()` except instead of returning -1, it raises a `ValueError` when the substring is not found. Equivalent to standard `str.index()`.

**Parameters**

**sub** [str] Substring being searched.

**start** [int] Left edge index.

**end** [int] Right edge index.

**Returns**

**Series of int** Series of lowest matching indexes.

**Examples**

```
>>> s = ks.Series(['apple', 'oranges', 'bananas'])
```

```
>>> s.str.index('a')
0    0
1    2
2    1
dtype: int64
```

The following expression throws an exception:

```
>>> s.str.index('a', start=2)
```

**databricks.koalas.Series.str.isalnum**

`str.isalnum() → ks.Series`

Check whether all characters in each string are alphanumeric.

This is equivalent to running the Python string method `str.isalnum()` for each element of the Series/Index. If a string has zero characters, `False` is returned for that check.

## Examples

```
>>> s1 = ks.Series(['one', 'one1', '1', ''])
```

```
>>> s1.str.isalnum()
0      True
1      True
2      True
3     False
dtype: bool
```

Note that checks against characters mixed with any additional punctuation or whitespace will evaluate to false for an alphanumeric check.

```
>>> s2 = ks.Series(['A B', '1.5', '3,000'])
>>> s2.str.isalnum()
0     False
1     False
2     False
dtype: bool
```

## **databricks.koalas.Series.str.isalpha**

**str.isalpha()** → ks.Series

Check whether all characters in each string are alphabetic.

This is equivalent to running the Python string method `str.isalpha()` for each element of the Series/Index. If a string has zero characters, False is returned for that check.

## Examples

```
>>> s1 = ks.Series(['one', 'one1', '1', ''])
```

```
>>> s1.str.isalpha()
0      True
1     False
2     False
3     False
dtype: bool
```

## **databricks.koalas.Series.str.isdigit**

**str.isdigit()** → ks.Series

Check whether all characters in each string are digits.

This is equivalent to running the Python string method `str.isdigit()` for each element of the Series/Index. If a string has zero characters, False is returned for that check.

## Examples

```
>>> s = ks.Series(['23', '3', '', ''])
```

The `s.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s.str.isdecimal()
0      True
1     False
2     False
3     False
dtype: bool
```

The `s.str.isdigit` method is the same as `s.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s.str.isdigit()
0      True
1      True
2     False
3     False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s.str.isnumeric()
0      True
1      True
2      True
3     False
dtype: bool
```

## `databricks.koalas.Series.str.isspace`

`str.isspace()` → `ks.Series`

Check whether all characters in each string are whitespaces.

This is equivalent to running the Python string method `str.isspace()` for each element of the Series/Index. If a string has zero characters, False is returned for that check.

## Examples

```
>>> s = ks.Series([' ', '\t\r\n ', ''])
>>> s.str.isspace()
0      True
1      True
2     False
dtype: bool
```

**databricks.koalas.Series.str.islower**

`str.islower()` → `ks.Series`

Check whether all characters in each string are lowercase.

This is equivalent to running the Python string method `str.islower()` for each element of the Series/Index. If a string has zero characters, False is returned for that check.

**Examples**

```
>>> s = ks.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
>>> s.str.islower()
0      True
1     False
2     False
3     False
dtype: bool
```

**databricks.koalas.Series.str.isupper**

`str.isupper()` → `ks.Series`

Check whether all characters in each string are uppercase.

This is equivalent to running the Python string method `str.isupper()` for each element of the Series/Index. If a string has zero characters, False is returned for that check.

**Examples**

```
>>> s = ks.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
>>> s.str.isupper()
0     False
1     False
2      True
3     False
dtype: bool
```

**databricks.koalas.Series.str.istitle**

`str.istitle()` → `ks.Series`

Check whether all characters in each string are titlecase.

This is equivalent to running the Python string method `str.istitle()` for each element of the Series/Index. If a string has zero characters, False is returned for that check.

## Examples

```
>>> s = ks.Series(['leopard', 'Golden Eagle', 'SNAKE', ''])
```

The `s.str.istitle` method checks for whether all words are in title case (whether only the first letter of each word is capitalized). Words are assumed to be as any sequence of non-numeric characters separated by whitespace characters.

```
>>> s.str.istitle()
0    False
1     True
2    False
3    False
dtype: bool
```

## databricks.koalas.Series.str.isnumeric

`str.isnumeric()` → `ks.Series`

Check whether all characters in each string are numeric.

This is equivalent to running the Python string method `str.isnumeric()` for each element of the Series/Index. If a string has zero characters, False is returned for that check.

## Examples

```
>>> s1 = ks.Series(['one', 'one1', '1', ''])
>>> s1.str.isnumeric()
0    False
1    False
2     True
3    False
dtype: bool
```

```
>>> s2 = ks.Series(['23', '3', '', ''])
```

The `s2.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s2.str.isdecimal()
0     True
1    False
2    False
3    False
dtype: bool
```

The `s2.str.isdigit` method is the same as `s2.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s2.str.isdigit()
0     True
1     True
2    False
3    False
dtype: bool
```

The `s2.str.isnumeric` method is the same as `s2.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s2.str.isnumeric()
0      True
1      True
2      True
3     False
dtype: bool
```

### **databricks.koalas.Series.str.isdecimal**

`str.isdecimal()` → `ks.Series`

Check whether all characters in each string are decimals.

This is equivalent to running the Python string method `str.isdecimal()` for each element of the Series/Index. If a string has zero characters, False is returned for that check.

### **Examples**

```
>>> s = ks.Series(['23', '³', '', ''])
```

The `s.str.isdecimal` method checks for characters used to form numbers in base 10.

```
>>> s.str.isdecimal()
0      True
1     False
2     False
3     False
dtype: bool
```

The `s.str.isdigit` method is the same as `s.str.isdecimal` but also includes special digits, like superscripted and subscripted digits in unicode.

```
>>> s.str.isdigit()
0      True
1      True
2     False
3     False
dtype: bool
```

The `s.str.isnumeric` method is the same as `s.str.isdigit` but also includes other characters that can represent quantities such as unicode fractions.

```
>>> s.str.isnumeric()
0      True
1      True
2      True
3     False
dtype: bool
```

**databricks.koalas.Series.str.join**`str.join(sep) → ks.Series`

Join lists contained as elements in the Series with passed delimiter.

If the elements of a Series are lists themselves, join the content of these lists using the delimiter passed to the function. This function is an equivalent to calling `str.join()` on the lists.

**Parameters****sep** [str] Delimiter to use between list entries.**Returns****Series of object** Series with list entries concatenated by intervening occurrences of the delimiter.**See also:****`str.split`** Split strings around given separator/delimiter.**`str.rsplit`** Splits string around given separator/delimiter, starting from the right.**Examples**

Example with a list that contains a None element.

```
>>> s = ks.Series([[ 'lion', 'elephant', 'zebra'],
...                [ 'cat', None, 'dog']])
>>> s
0    [lion, elephant, zebra]
1    [cat, None, dog]
dtype: object
```

Join all lists using a '-'. The list containing None will produce None.

```
>>> s.str.join('-')
0    lion-elephant-zebra
1                      None
dtype: object
```

**databricks.koalas.Series.str.len**`str.len() → ks.Series`

Computes the length of each element in the Series.

The element may be a sequence (such as a string, tuple or list).

**Returns****Series of int** A Series of integer values indicating the length of each element in the Series.

## Examples

Returns the length (number of characters) in a string. Returns the number of entries for lists or tuples.

```
>>> s1 = ks.Series(['dog', 'monkey'])
>>> s1.str.len()
0      3
1      6
dtype: int64
```

```
>>> s2 = ks.Series(["a", "b", "c"], [])
>>> s2.str.len()
0      3
1      0
dtype: int64
```

## `databricks.koalas.Series.str.ljust`

`str.ljust(width, fillchar=' ')` → `ks.Series`

Filling right side of strings in the Series with an additional character. Equivalent to `str.ljust()`.

### Parameters

**width** [int] Minimum width of resulting string; additional characters will be filled with *fillchar*.

**fillchar** [str] Additional character for filling, default is whitespace.

### Returns

Series of object

## Examples

```
>>> s = ks.Series(["caribou", "tiger"])
>>> s
0      caribou
1        tiger
dtype: object
```

```
>>> s.str.ljust(width=10, fillchar='-')
0      caribou---
1      tiger-----
dtype: object
```

## `databricks.koalas.Series.str.lower`

`str.lower()` → `ks.Series`

Convert strings in the Series/Index to all lowercase.



## Examples

```
>>> s = ks.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0          lower
1        CAPITALS
2    this is a sentence
3        SwApCaSe
dtype: object
```

```
>>> s.str.lower()
0          lower
1        capitals
2    this is a sentence
3        swapcase
dtype: object
```

## databricks.koalas.Series.str.lstrip

`str.lstrip(to_strip=None) → ks.Series`

Remove leading characters.

Strip whitespaces (including newlines) or a set of specified characters from each string in the Series/Index from left side. Equivalent to `str.lstrip()`.

### Parameters

**to\_strip** [str] Specifying the set of characters to be removed. All combinations of this set of characters will be stripped. If None then whitespaces are removed.

### Returns

**Series of object**

## Examples

```
>>> s = ks.Series(['1. Ant.', '2. Bee!\t', None])
>>> s
0    1. Ant.
1    2. Bee!\t
2         None
dtype: object
```

```
>>> s.str.lstrip('12.')
0    Ant.
1    Bee!\t
2         None
dtype: object
```

**databricks.koalas.Series.str.match**

`str.match(pat, case=True, flags=0, na=nan) → ks.Series`

Determine if each string matches a regular expression.

Analogous to `contains()`, but more strict, relying on `re.match()` instead of `re.search()`.

**Parameters**

**pat** [str] Character sequence or regular expression.

**case** [bool, default True] If True, case sensitive.

**flags** [int, default 0 (no flags)] Flags to pass through to the re module, e.g. `re.IGNORECASE`.

**na** [default NaN] Fill value for missing values.

**Returns**

**Series of boolean values or object** A Series of boolean values indicating whether the given pattern can be matched in the string of each element of the Series.

**Examples**

```
>>> s = ks.Series(['Mouse', 'dog', 'house and parrot', '23', np.NaN])
>>> s.str.match('dog')
0    False
1     True
2    False
3    False
4     None
dtype: object
```

```
>>> s.str.match('mouse|dog', case=False)
0     True
1     True
2    False
3    False
4     None
dtype: object
```

```
>>> s.str.match('.+and.+', na=True)
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

```
>>> import re
>>> s.str.match('MOUSE', flags=re.IGNORECASE)
0     True
1    False
2    False
3    False
4     None
dtype: object
```

**databricks.koalas.Series.str.normalize**`str.normalize(form) → ks.Series`

Return the Unicode normal form for the strings in the Series.

For more information on the forms, see the `unicodedata.normalize()`.**Parameters****form** [{‘NFC’, ‘NFKC’, ‘NFD’, ‘NFKD’}] Unicode form.**Returns****Series of objects** A Series of normalized strings.**databricks.koalas.Series.str.pad**`str.pad(width, side='left', fillchar=' ') → ks.Series`

Pad strings in the Series up to width.

**Parameters****width** [int] Minimum width of resulting string; additional characters will be filled with character defined in *fillchar*.**side** [{‘left’, ‘right’, ‘both’}, default ‘left’] Side from which to fill resulting string.**fillchar** [str, default ‘ ’] Additional character for filling, default is whitespace.**Returns****Series of object** Returns Series with minimum number of char in object.**Examples**

```
>>> s = ks.Series(["caribou", "tiger"])
>>> s
0    caribou
1     tiger
dtype: object
```

```
>>> s.str.pad(width=10)
0    caribou
1     tiger
dtype: object
```

```
>>> s.str.pad(width=10, side='right', fillchar='-')
0    caribou---
1    tiger----
dtype: object
```

```
>>> s.str.pad(width=10, side='both', fillchar='-')
0    -caribou--
1    --tiger---
dtype: object
```

**databricks.koalas.Series.str.partition**

`str.partition` (*sep*=' ', *expand*=True) → ks.Series  
Not supported.

**databricks.koalas.Series.str.repeat**

`str.repeat` (*repeats*) → ks.Series  
Duplicate each string in the Series.

**Parameters**

**repeats** [int] Repeat the string given number of times (int). Sequence of int is not supported.

**Returns**

**Series of object** Series or Index of repeated string objects specified by input parameter repeats.

**Examples**

```
>>> s = ks.Series(['a', 'b', 'c'])
>>> s
0    a
1    b
2    c
dtype: object
```

Single int repeats string in Series

```
>>> s.str.repeat(repeats=2)
0    aa
1    bb
2    cc
dtype: object
```

**databricks.koalas.Series.str.replace**

`str.replace` (*pat*, *repl*, *n*=-1, *case*=None, *flags*=0, *regex*=True) → ks.Series  
Replace occurrences of pattern/regex in the Series with some other string. Equivalent to `str.replace()` or `re.sub()`.

**Parameters**

**pat** [str or compiled regex] String can be a character sequence or regular expression.

**repl** [str or callable] Replacement string or a callable. The callable is passed the regex match object and must return a replacement string to be used. See `re.sub()`.

**n** [int, default -1 (all)] Number of replacements to make from start.

**case** [boolean, default None] If True, case sensitive (the default if pat is a string). Set to False for case insensitive. Cannot be set if pat is a compiled regex.

**flags: int, default 0 (no flags)** re module flags, e.g. `re.IGNORECASE`. Cannot be set if pat is a compiled regex.

**regex** [boolean, default True] If True, assumes the passed-in pattern is a regular expression. If False, treats the pattern as a literal string. Cannot be set to False if pat is a compile regex or repl is a callable.

### Returns

**Series of object** A copy of the string with all matching occurrences of pat replaced by repl.

### Examples

When pat is a string and regex is True (the default), the given pat is compiled as a regex. When repl is a string, it replaces matching regex patterns as with `re.sub()`. NaN value(s) in the Series are changed to None:

```
>>> ks.Series(['foo', 'fuz', np.nan]).str.replace('f.', 'ba', regex=True)
0    bao
1    baz
2    None
dtype: object
```

When pat is a string and regex is False, every pat is replaced with repl as with `str.replace()`:

```
>>> ks.Series(['f.o', 'fuz', np.nan]).str.replace('f.', 'ba', regex=False)
0    bao
1    fuz
2    None
dtype: object
```

When repl is a callable, it is called on every pat using `re.sub()`. The callable should expect one positional argument (a regex object) and return a string.

Reverse every lowercase alphabetic word:

```
>>> repl = lambda m: m.group(0)[::-1]
>>> ks.Series(['foo 123', 'bar baz', np.nan]).str.replace(r'[a-z]+', repl)
0    oof 123
1    rab zab
2    None
dtype: object
```

Using regex groups (extract second group and swap case):

```
>>> pat = r"(?P<one>\w+) (?P<two>\w+) (?P<three>\w+)"
>>> repl = lambda m: m.group('two').swapcase()
>>> ks.Series(['One Two Three', 'Foo Bar Baz']).str.replace(pat, repl)
0    tWO
1    bAR
dtype: object
```

Using a compiled regex with flags:

```
>>> import re
>>> regex_pat = re.compile(r'FUZ', flags=re.IGNORECASE)
>>> ks.Series(['foo', 'fuz', np.nan]).str.replace(regex_pat, 'bar')
0    foo
1    bar
2    None
dtype: object
```

**databricks.koalas.Series.str.rfind**

`str.rfind(sub, start=0, end=None) → ks.Series`

Return highest indexes in each strings in the Series where the substring is fully contained between [start:end].

Return -1 on failure. Equivalent to standard `str.rfind()`.

**Parameters**

**sub** [str] Substring being searched.

**start** [int] Left edge index.

**end** [int] Right edge index.

**Returns**

**Series of int** Series of highest matching indexes.

**Examples**

```
>>> s = ks.Series(['apple', 'oranges', 'bananas'])
```

```
>>> s.str.rfind('a')
0      0
1      2
2      5
dtype: int64
```

```
>>> s.str.rfind('a', start=2)
0     -1
1      2
2      5
dtype: int64
```

```
>>> s.str.rfind('a', end=1)
0      0
1     -1
2     -1
dtype: int64
```

```
>>> s.str.rfind('a', start=2, end=2)
0     -1
1     -1
2     -1
dtype: int64
```

**databricks.koalas.Series.str.rindex**

`str.rindex(sub, start=0, end=None) → ks.Series`

Return highest indexes in each strings where the substring is fully contained between [start:end].

This is the same as `str.rfind()` except instead of returning -1, it raises a `ValueError` when the substring is not found. Equivalent to standard `str.rindex()`.

**Parameters**

**sub** [str] Substring being searched.

**start** [int] Left edge index.

**end** [int] Right edge index.

**Returns**

**Series of int** Series of highest matching indexes.

**Examples**

```
>>> s = ks.Series(['apple', 'oranges', 'bananas'])
```

```
>>> s.str.rindex('a')
0      0
1      2
2      5
dtype: int64
```

The following expression throws an exception:

```
>>> s.str.rindex('a', start=2)
```

**databricks.koalas.Series.str.rjust**

`str.rjust(width, fillchar=' ') → ks.Series`

Filling left side of strings in the Series with an additional character. Equivalent to `str.rjust()`.

**Parameters**

**width** [int] Minimum width of resulting string; additional characters will be filled with *fillchar*.

**fillchar** [str] Additional character for filling, default is whitespace.

**Returns**

**Series of object**

## Examples

```
>>> s = ks.Series(["caribou", "tiger"])
>>> s
0    caribou
1     tiger
dtype: object
```

```
>>> s.str.rjust(width=10)
0    caribou
1     tiger
dtype: object
```

```
>>> s.str.rjust(width=10, fillchar='-')
0    ---caribou
1    ----tiger
dtype: object
```

## `databricks.koalas.Series.str.rpartition`

`str.rpartition(sep=' ', expand=True) → ks.Series`  
Not supported.

## `databricks.koalas.Series.str.rsplit`

`str.rsplit(pat=None, n=-1, expand=False) → Union[ks.Series, ks.DataFrame]`  
Split strings around given separator/delimiter.

Splits the string in the Series from the end, at the specified delimiter string. Equivalent to `str.rsplit()`.

### Parameters

**pat** [str, optional] String or regular expression to split on. If not specified, split on whitespace.

**n** [int, default -1 (all)] Limit number of splits in output. None, 0 and -1 will be interpreted as return all splits.

**expand** [bool, default False] Expand the splitted strings into separate columns.

- If `True`, `n` must be a positive integer, and return `DataFrame` expanding dimensionality.
- If `False`, return `Series`, containing lists of strings.

### Returns

**Series, DataFrame** Type matches caller unless `expand=True` (see Notes).

See also:

`str.split` Split strings around given separator/delimiter.

`str.join` Join lists contained as elements in the Series/Index with passed delimiter.



## Notes

The handling of the  $n$  keyword depends on the number of found splits:

- If found splits  $> n$ , make first  $n$  splits only
- If found splits  $\leq n$ , make all splits
- If for a certain row the number of found splits  $< n$ , append *None* for padding up to  $n$  if `expand=True`

If using `expand=True`, Series callers return DataFrame objects with  $n + 1$  columns.

---

**Note:** Even if  $n$  is much larger than found splits, the number of columns does NOT shrink unlike pandas.

---

## Examples

```
>>> s = ks.Series(["this is a regular sentence",
...               "https://docs.python.org/3/tutorial/index.html",
...               np.nan])
```

In the default setting, the string is split by whitespace.

```
>>> s.str.split()
0      [this, is, a, regular, sentence]
1      [https://docs.python.org/3/tutorial/index.html]
2      None
dtype: object
```

Without the  $n$  parameter, the outputs of `rsplit` and `split` are identical.

```
>>> s.str.rsplit()
0      [this, is, a, regular, sentence]
1      [https://docs.python.org/3/tutorial/index.html]
2      None
dtype: object
```

The  $n$  parameter can be used to limit the number of splits on the delimiter. The outputs of `split` and `rsplit` are different.

```
>>> s.str.split(n=2)
0      [this, is, a regular sentence]
1      [https://docs.python.org/3/tutorial/index.html]
2      None
dtype: object
```

```
>>> s.str.rsplit(n=2)
0      [this is a, regular, sentence]
1      [https://docs.python.org/3/tutorial/index.html]
2      None
dtype: object
```

When using `expand=True`, the split elements will expand out into separate columns. If NaN is present, it is propagated throughout the columns during the split.

```
>>> s.str.split(n=4, expand=True)
```

	0	1	2	3	4
0	this	is	a	regular	sentence
1	https://docs.python.org/3/tutorial/index.html	None	None	None	None
2	None	None	None	None	None

For slightly more complex use cases like splitting the html document name from a url, a combination of parameter settings can be used.

```
>>> s.str.rsplit("/", n=1, expand=True)
```

	0	1
0	this is a regular sentence	None
1	https://docs.python.org/3/tutorial	index.html
2	None	None

Remember to escape special characters when explicitly using regular expressions.

```
>>> s = ks.Series(["1+1=2"])
>>> s.str.split(r"\+|= ", n=2, expand=True)
```

	0	1	2
0	1	1	2

## databricks.koalas.Series.str.rstrip

`str.rstrip(to_strip=None) → ks.Series`

Remove trailing characters.

Strip whitespaces (including newlines) or a set of specified characters from each string in the Series/Index from right side. Equivalent to `str.rstrip()`.

### Parameters

**to\_strip** [str] Specifying the set of characters to be removed. All combinations of this set of characters will be stripped. If None then whitespaces are removed.

### Returns

Series of object

## Examples

```
>>> s = ks.Series(['1. Ant.', '2. Bee!\t', None])
>>> s
```

0	1. Ant.
1	2. Bee!\t
2	None

dtype: object

```
>>> s.str.rstrip('!\t')
```

0	1. Ant
1	2. Bee
2	None

dtype: object

**databricks.koalas.Series.str.slice**

`str.slice` (*start=None, stop=None, step=None*) → `ks.Series`  
 Slice substrings from each element in the Series.

**Parameters**

- start** [int, optional] Start position for slice operation.
- stop** [int, optional] Stop position for slice operation.
- step** [int, optional] Step size for slice operation.

**Returns**

**Series of object** Series from sliced substrings from original string objects.

**Examples**

```
>>> s = ks.Series(["koala", "fox", "chameleon"])
>>> s
0      koala
1       fox
2  chameleon
dtype: object
```

```
>>> s.str.slice(start=1)
0      oala
1       ox
2  hameleon
dtype: object
```

```
>>> s.str.slice(stop=2)
0      ko
1      fo
2      ch
dtype: object
```

```
>>> s.str.slice(step=2)
0      kaa
1       fx
2    caeen
dtype: object
```

```
>>> s.str.slice(start=0, stop=5, step=3)
0      kl
1       f
2      cm
dtype: object
```

**databricks.koalas.Series.str.slice\_replace**

`str.slice_replace(start=None, stop=None, repl=None) → ks.Series`

Slice substrings from each element in the Series.

**Parameters**

**start** [int, optional] Start position for slice operation. If not specified (None), the slice is unbounded on the left, i.e. slice from the start of the string.

**stop** [int, optional] Stop position for slice operation. If not specified (None), the slice is unbounded on the right, i.e. slice until the end of the string.

**repl** [str, optional] String for replacement. If not specified (None), the sliced region is replaced with an empty string.

**Returns**

**Series of object** Series from sliced substrings from original string objects.

**Examples**

```
>>> s = ks.Series(['a', 'ab', 'abc', 'abdc', 'abcde'])
>>> s
0      a
1     ab
2    abc
3   abdc
4  abcde
dtype: object
```

Specify just start, meaning replace start until the end of the string with repl.

```
>>> s.str.slice_replace(1, repl='X')
0    aX
1    aX
2    aX
3    aX
4    aX
dtype: object
```

Specify just stop, meaning the start of the string to stop is replaced with repl, and the rest of the string is included.

```
>>> s.str.slice_replace(stop=2, repl='X')
0      X
1      X
2     Xc
3    Xdc
4   Xcde
dtype: object
```

Specify start and stop, meaning the slice from start to stop is replaced with repl. Everything before or after start and stop is included as is.

```
>>> s.str.slice_replace(start=1, stop=3, repl='X')
0    aX
1    aX
2    aX
```

(continues on next page)

(continued from previous page)

```

3      aXc
4      aXde
dtype: object

```

**databricks.koalas.Series.str.split**

`str.split` (*pat=None, n=-1, expand=False*) → Union[ks.Series, ks.DataFrame]

Split strings around given separator/delimiter.

Splits the string in the Series from the beginning, at the specified delimiter string. Equivalent to `str.split()`.

**Parameters**

**pat** [str, optional] String or regular expression to split on. If not specified, split on whitespace.

**n** [int, default -1 (all)] Limit number of splits in output. None, 0 and -1 will be interpreted as return all splits.

**expand** [bool, default False] Expand the splitted strings into separate columns.

- If `True`, *n* must be a positive integer, and return DataFrame expanding dimensionality.
- If `False`, return Series, containing lists of strings.

**Returns**

**Series, DataFrame** Type matches caller unless `expand=True` (see Notes).

See also:

`str.rsplit` Splits string around given separator/delimiter, starting from the right.

`str.join` Join lists contained as elements in the Series/Index with passed delimiter.

**Notes**

The handling of the *n* keyword depends on the number of found splits:

- If found splits > *n*, make first *n* splits only
- If found splits ≤ *n*, make all splits
- If for a certain row the number of found splits < *n*, append *None* for padding up to *n* if `expand=True`

If using `expand=True`, Series callers return DataFrame objects with *n + 1* columns.

---

**Note:** Even if *n* is much larger than found splits, the number of columns does NOT shrink unlike pandas.

---

## Examples

```
>>> s = ks.Series(["this is a regular sentence",
...               "https://docs.python.org/3/tutorial/index.html",
...               np.nan])
```

In the default setting, the string is split by whitespace.

```
>>> s.str.split()
0          [this, is, a, regular, sentence]
1    [https://docs.python.org/3/tutorial/index.html]
2                                     None
dtype: object
```

Without the `n` parameter, the outputs of `rsplit` and `split` are identical.

```
>>> s.str.rsplit()
0          [this, is, a, regular, sentence]
1    [https://docs.python.org/3/tutorial/index.html]
2                                     None
dtype: object
```

The `n` parameter can be used to limit the number of splits on the delimiter. The outputs of `split` and `rsplit` are different.

```
>>> s.str.split(n=2)
0          [this, is, a regular sentence]
1    [https://docs.python.org/3/tutorial/index.html]
2                                     None
dtype: object
```

```
>>> s.str.rsplit(n=2)
0          [this is a, regular, sentence]
1    [https://docs.python.org/3/tutorial/index.html]
2                                     None
dtype: object
```

The `pat` parameter can be used to split by other characters.

```
>>> s.str.split(pat = "/")
0          [this is a regular sentence]
1    [https:, , docs.python.org, 3, tutorial, index...]
2                                     None
dtype: object
```

When using `expand=True`, the split elements will expand out into separate columns. If `NaN` is present, it is propagated throughout the columns during the split.

```
>>> s.str.split(n=4, expand=True)
```

	0	1	2	3	4
0	this	is	a	regular	sentence
1	https://docs.python.org/3/tutorial/index.html	None	None	None	None
2	None	None	None	None	None

For slightly more complex use cases like splitting the html document name from a url, a combination of parameter settings can be used.

```
>>> s.str.rsplit("/", n=1, expand=True)
      0      1
0      this is a regular sentence      None
1  https://docs.python.org/3/tutorial  index.html
2      None      None
```

Remember to escape special characters when explicitly using regular expressions.

```
>>> s = ks.Series(["1+1=2"])
>>> s.str.split(r"\+|= ", n=2, expand=True)
      0  1  2
0  1  1  2
```

### databricks.koalas.Series.str.startswith

`str.startswith(pattern, na=None) → ks.Series`

Test if the start of each string element matches a pattern.

Equivalent to `str.startswith()`.

#### Parameters

**pattern** [str] Character sequence. Regular expressions are not accepted.

**na** [object, default None] Object shown if element is not a string. NaN converted to None.

#### Returns

**Series of bool or object** Koalas Series of booleans indicating whether the given pattern matches the start of each string element.

### Examples

```
>>> s = ks.Series(['bat', 'Bear', 'cat', np.nan])
>>> s
0      bat
1      Bear
2      cat
3      None
dtype: object
```

```
>>> s.str.startswith('b')
0      True
1     False
2     False
3      None
dtype: object
```

Specifying na to be False instead of None.

```
>>> s.str.startswith('b', na=False)
0      True
1     False
2     False
3     False
dtype: bool
```

**databricks.koalas.Series.str.strip**

`str.strip(to_strip=None) → ks.Series`  
Remove leading and trailing characters.

Strip whitespaces (including newlines) or a set of specified characters from each string in the Series/Index from left and right sides. Equivalent to `str.strip()`.

**Parameters**

**to\_strip** [str] Specifying the set of characters to be removed. All combinations of this set of characters will be stripped. If None then whitespaces are removed.

**Returns**

**Series of objects**

**Examples**

```
>>> s = ks.Series(['1. Ant.', '2. Bee!\t', None])
>>> s
0      1. Ant.
1      2. Bee!\t
2      None
dtype: object
```

```
>>> s.str.strip()
0      1. Ant.
1      2. Bee!
2      None
dtype: object
```

```
>>> s.str.strip('12.')
0      Ant
1      Bee!\t
2      None
dtype: object
```

```
>>> s.str.strip('!\t')
0      1. Ant
1      2. Bee
2      None
dtype: object
```

**databricks.koalas.Series.str.swapcase**

`str.swapcase() → ks.Series`  
Convert strings in the Series/Index to be swapcased.



### Examples

```
>>> s = ks.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0          lower
1        CAPITALS
2    this is a sentence
3          SwApCaSe
dtype: object
```

```
>>> s.str.swapcase()
0          LOWER
1        capitals
2    THIS IS A SENTENCE
3          sWaPcAsE
dtype: object
```

### `databricks.koalas.Series.str.title`

`str.title()` → `ks.Series`  
Convert Strings in the series to be titlecase.

### Examples

```
>>> s = ks.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0          lower
1        CAPITALS
2    this is a sentence
3          SwApCaSe
dtype: object
```

```
>>> s.str.title()
0          Lower
1        Capitals
2    This Is A Sentence
3          Swapcase
dtype: object
```

### `databricks.koalas.Series.str.translate`

`str.translate(table)` → `ks.Series`  
Map all characters in the string through the given mapping table. Equivalent to standard `str.translate()`.

#### Parameters

**table** [dict] Table is a mapping of Unicode ordinals to Unicode ordinals, strings, or None. Unmapped characters are left untouched. Characters mapped to None are deleted. `str.maketrans()` is a helper function for making translation tables.

#### Returns

**Series of object** Series with translated strings.

### Examples

```
>>> s = ks.Series(["dog", "cat", "bird"])
>>> m = str.maketrans({'a': 'X', 'i': 'Y', 'o': None})
>>> s.str.translate(m)
0      dg
1     cXt
2    bYrd
dtype: object
```

### `databricks.koalas.Series.str.upper`

`str.upper()` → `ks.Series`

Convert strings in the Series/Index to all uppercase.

### Examples

```
>>> s = ks.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0          lower
1        CAPITALS
2  this is a sentence
3        SwApCaSe
dtype: object
```

```
>>> s.str.upper()
0          LOWER
1        CAPITALS
2  THIS IS A SENTENCE
3        SWAPCASE
dtype: object
```

### `databricks.koalas.Series.str.wrap`

`str.wrap(width, **kwargs)` → `ks.Series`

Wrap long strings in the Series to be formatted in paragraphs with length less than a given width.

This method has the same keyword parameters and defaults as `textwrap.TextWrapper`.

#### Parameters

**width** [int] Maximum line-width. Lines separated with newline char.

**expand\_tabs** [bool, optional] If true, tab characters will be expanded to spaces (default: True).

**replace\_whitespace** [bool, optional] If true, each whitespace character remaining after tab expansion will be replaced by a single space (default: True).

**drop\_whitespace** [bool, optional] If true, whitespace that, after wrapping, happens to end up at the beginning or end of a line is dropped (default: True).

**break\_long\_words** [bool, optional] If true, then words longer than width will be broken in order to ensure that no lines are longer than width. If it is false, long words will not be broken, and some lines may be longer than width (default: True).

**break\_on\_hyphens** [bool, optional] If true, wrapping will occur preferably on whitespace and right after hyphens in compound words, as it is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks, but you need to set `break_long_words` to false if you want truly insecable words (default: True).

### Returns

**Series of object** Series with wrapped strings.

### Examples

```
>>> s = ks.Series(['line to be wrapped', 'another line to be wrapped'])
>>> s.str.wrap(12)
0          line to be\nwrapped
1  another line\nto be\nwrapped
dtype: object
```

## databricks.koalas.Series.str.zfill

`str.zfill(width) → ks.Series`

Pad strings in the Series by prepending '0' characters.

Strings in the Series are padded with '0' characters on the left of the string to reach a total string length width. Strings in the Series with length greater or equal to width are unchanged.

Differs from `str.zfill()` which has special handling for '+' '-' in the string.

### Parameters

**width** [int] Minimum length of resulting string; strings with length less than width be prepended with '0' characters.

### Returns

**Series of object** Series with '0' left-padded strings.

### Examples

```
>>> s = ks.Series(['-1', '1', '1000', np.nan])
>>> s
0      -1
1       1
2    1000
3     None
dtype: object
```

Note that NaN is not a string, therefore it is converted to NaN. The minus sign in '-1' is treated as a regular character and the zero is added to the left of it (`str.zfill()` would have moved it to the left). 1000 remains unchanged as it is longer than width.

```
>>> s.str.zfill(3)
0     0-1
1     001
2    1000
3     None
dtype: object
```

### 3.3.17 Categorical accessor

Categorical-dtype specific methods and attributes are available under the `Series.cat` accessor.

<code>Series.cat.categories</code>	The categories of this categorical.
<code>Series.cat.ordered</code>	Whether the categories have an ordered relationship.
<code>Series.cat.codes</code>	Return Series of codes as well as the index.

#### `databricks.koalas.Series.cat.categories`

**property** `cat.categories`

The categories of this categorical.

#### Examples

```
>>> s = ks.Series(list("abbccc"), dtype="category")
>>> s
0    a
1    b
2    b
3    c
4    c
5    c
dtype: category
Categories (3, object): ['a', 'b', 'c']
```

```
>>> s.cat.categories
Index(['a', 'b', 'c'], dtype='object')
```

#### `databricks.koalas.Series.cat.ordered`

**property** `cat.ordered`

Whether the categories have an ordered relationship.

#### Examples

```
>>> s = ks.Series(list("abbccc"), dtype="category")
>>> s
0    a
1    b
2    b
3    c
4    c
5    c
dtype: category
Categories (3, object): ['a', 'b', 'c']
```

```
>>> s.cat.ordered
False
```

**databricks.koalas.Series.cat.codes****property** `cat.codes`

Return Series of codes as well as the index.

**Examples**

```
>>> s = ks.Series(list("abbccc"), dtype="category")
>>> s
0    a
1    b
2    b
3    c
4    c
5    c
dtype: category
Categories (3, object): ['a', 'b', 'c']
```

```
>>> s.cat.codes
0    0
1    1
2    1
3    2
4    2
5    2
dtype: int8
```

**3.3.18 Plotting**

`Series.plot` is both a callable method and a namespace attribute for specific plotting methods of the form `Series.plot.<kind>`.

<code>Series.plot</code>	alias of <code>databricks.koalas.plot.core.KoalasPlotAccessor</code>
<code>Series.plot.area([x, y])</code>	Draw a stacked area plot.
<code>Series.plot.bar([x, y])</code>	Vertical bar plot.
<code>Series.plot.barh([x, y])</code>	Make a horizontal bar plot.
<code>Series.plot.box(**kws)</code>	Make a box plot of the Series columns.
<code>Series.plot.density([bw_method, ind])</code>	Generate Kernel Density Estimate plot using Gaussian kernels.
<code>Series.plot.hist([bins])</code>	Draw one histogram of the DataFrame's columns.
<code>Series.plot.line([x, y])</code>	Plot DataFrame/Series as lines.
<code>Series.plot.pie(**kws)</code>	Generate a pie plot.
<code>Series.plot.kde([bw_method, ind])</code>	Generate Kernel Density Estimate plot using Gaussian kernels.
<code>Series.hist([bins])</code>	Draw one histogram of the DataFrame's columns.

## **databricks.koalas.Series.plot**

**databricks.koalas.Series.plot**

alias of `databricks.koalas.plot.core.KoalasPlotAccessor`

## **databricks.koalas.Series.plot.area**

`plot.area` (*x=None*, *y=None*, *\*\*kws*)

Draw a stacked area plot.

An area plot displays quantitative data visually. This function wraps the plotly area function.

### **Parameters**

**x** [label or position, optional] Coordinates for the X axis. By default uses the index.

**y** [label or position, optional] Column to plot. By default uses all columns.

**stacked** [bool, default True] Area plots are stacked by default. Set to False to create a unstacked plot (matplotlib-only).

**\*\*kws** [optional] Additional keyword arguments are documented in `DataFrame.plot()`.

### **Returns**

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

## **Examples**

For Series

```
>>> df = ks.DataFrame({
...     'sales': [3, 2, 3, 9, 10, 6],
...     'signups': [5, 5, 6, 12, 14, 13],
...     'visits': [20, 42, 28, 62, 81, 50],
... }, index=pd.date_range(start='2018/01/01', end='2018/07/01',
...                          freq='M'))
>>> df.sales.plot.area()
```

For DataFrame

```
>>> df = ks.DataFrame({
...     'sales': [3, 2, 3, 9, 10, 6],
...     'signups': [5, 5, 6, 12, 14, 13],
...     'visits': [20, 42, 28, 62, 81, 50],
... }, index=pd.date_range(start='2018/01/01', end='2018/07/01',
...                          freq='M'))
>>> df.plot.area()
```

**databricks.koalas.Series.plot.bar**

`plot.bar` ( $x=None$ ,  $y=None$ ,  $**kws$ )  
Vertical bar plot.

**Parameters**

- x** [label or position, optional] Allows plotting of one column versus another. If not specified, the index of the DataFrame is used.
- y** [label or position, optional] Allows plotting of one column versus another. If not specified, all numerical columns are used.
- \*\*kws** [optional] Additional keyword arguments are documented in `Koalas.Series.plot()` or `Koalas.DataFrame.plot()`.

**Returns**

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

**Examples**

Basic plot.

For Series:

```
>>> s = ks.Series([1, 3, 2])
>>> s.plot.bar()
```

For DataFrame:

```
>>> df = ks.DataFrame({'lab': ['A', 'B', 'C'], 'val': [10, 30, 20]})
>>> df.plot.bar(x='lab', y='val')
```

Plot a whole dataframe to a bar plot. Each column is stacked with a distinct color along the horizontal axis.

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> df.plot.bar()
```

Instead of stacking, the figure can be split by column with plotly APIs.

```
>>> from plotly.subplots import make_subplots
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> fig = (make_subplots(rows=2, cols=1)
...       .add_trace(df.plot.bar(y='speed').data[0], row=1, col=1)
...       .add_trace(df.plot.bar(y='speed').data[0], row=1, col=1)
...       .add_trace(df.plot.bar(y='lifespan').data[0], row=2, col=1))
>>> fig
```

Plot a single column.

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...          'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> df.plot.bar(y='speed')
```

Plot only selected categories for the DataFrame.

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...          'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> df.plot.bar(x='lifespan')
```

### **databricks.koalas.Series.plot.barh**

`plot.barh(x=None, y=None, **kwargs)`

Make a horizontal bar plot.

A horizontal bar plot is a plot that presents quantitative data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

#### **Parameters**

**x** [label or position, default DataFrame.index] Column to be used for categories.

**y** [label or position, default All numeric columns in dataframe] Columns to be plotted from the DataFrame.

**\*\*kwargs** Keyword arguments to pass on to `databricks.koalas.DataFrame.plot()` or `databricks.koalas.Series.plot()`.

#### **Returns**

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

See also:

**plotly.express.bar** Plot a vertical bar plot using plotly.

**matplotlib.axes.Axes.bar** Plot a vertical bar plot using matplotlib.



## Examples

For Series:

```
>>> df = ks.DataFrame({'lab': ['A', 'B', 'C'], 'val': [10, 30, 20]})
>>> df.val.plot.barh()
```

For DataFrame:

```
>>> df = ks.DataFrame({'lab': ['A', 'B', 'C'], 'val': [10, 30, 20]})
>>> df.plot.barh(x='lab', y='val')
```

Plot a whole DataFrame to a horizontal bar plot

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> df.plot.barh()
```

Plot a column of the DataFrame to a horizontal bar plot

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> df.plot.barh(y='speed')
```

Plot DataFrame versus the desired column

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> df.plot.barh(x='lifespan')
```

## `databricks.koalas.Series.plot.box`

`plot.box(**kws)`

Make a box plot of the Series columns.

### Parameters

**\*\*kws** [optional] Additional keyword arguments are documented in `Koalas.Series.plot()`.

**precision: scalar, default = 0.01** This argument is used by Koalas to compute approximate statistics for building a boxplot. Use *smaller* values to get more precise statistics (matplotlib-only).

### Returns

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

## Notes

There are behavior differences between Koalas and pandas.

- Koalas computes approximate statistics - expect differences between pandas and Koalas boxplots, especially regarding 1st and 3rd quartiles.
- The *whis* argument is only supported as a single number.
- Koalas doesn't support the following argument(s) (matplotlib-only).
  - *bootstrap* argument is not supported
  - *autorange* argument is not supported

## Examples

Draw a box plot from a DataFrame with four columns of randomly generated data.

For Series:

```
>>> data = np.random.randn(25, 4)
>>> df = ks.DataFrame(data, columns=list('ABCD'))
>>> df['A'].plot.box()
```

This is an unsupported function for DataFrame type

## databricks.koalas.Series.plot.density

**plot.density** (*bw\_method=None, ind=None, \*\*kwargs*)

Generate Kernel Density Estimate plot using Gaussian kernels.

### Parameters

**bw\_method** [scalar] The method used to calculate the estimator bandwidth. See `KernelDensity` in PySpark for more information.

**ind** [NumPy array or integer, optional] Evaluation points for the estimated PDF. If *None* (default), 1000 equally spaced points are used. If *ind* is a NumPy array, the KDE is evaluated at the points passed. If *ind* is an integer, *ind* number of equally spaced points are used.

**\*\*kwargs** [optional] Keyword arguments to pass on to `Koalas.Series.plot()`.

### Returns

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

## Examples

A scalar bandwidth should be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```
>>> s = ks.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> s.plot.kde(bw_method=0.3)
```

```
>>> s = ks.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> s.plot.kde(bw_method=3)
```

The *ind* parameter determines the evaluation points for the plot of the estimated KDF:

```
>>> s = ks.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> s.plot.kde(ind=[1, 2, 3, 4, 5], bw_method=0.3)
```

For DataFrame, it works in the same way as Series:

```
>>> df = ks.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> df.plot.kde(bw_method=0.3)
```

```
>>> df = ks.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> df.plot.kde(bw_method=3)
```

```
>>> df = ks.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> df.plot.kde(ind=[1, 2, 3, 4, 5, 6], bw_method=0.3)
```

## databricks.koalas.Series.plot.hist

`plot.hist(bins=10, **kws)`

Draw one histogram of the DataFrame's columns. A **histogram** is a representation of the distribution of data. This function calls `plotting.backend.plot()`, on each series in the DataFrame, resulting in one histogram per column.

### Parameters

**bins** [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

**\*\*kws** All other plotting keyword arguments to be passed to plotting backend.

### Returns

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

## Examples

Basic plot.

For Series:

```
>>> s = ks.Series([1, 3, 2])
>>> s.plot.hist()
```

For DataFrame:

```
>>> df = pd.DataFrame(
...     np.random.randint(1, 7, 6000),
...     columns=['one'])
>>> df['two'] = df['one'] + np.random.randint(1, 7, 6000)
>>> df = ks.from_pandas(df)
>>> df.plot.hist(bins=12, alpha=0.5)
```

## databricks.koalas.Series.plot.line

`plot.line` (*x=None*, *y=None*, *\*\*kwargs*)

Plot DataFrame/Series as lines.

This function is useful to plot lines using Series's values as coordinates.

### Parameters

- x** [int or str, optional] Columns to use for the horizontal axis. Either the location or the label of the columns to be used. By default, it will use the DataFrame indices.
- y** [int, str, or list of them, optional] The values to be plotted. Either the location or the label of the columns to be used. By default, it will use the remaining DataFrame numeric columns.
- \*\*kwargs** Keyword arguments to pass on to `Series.plot()` or `DataFrame.plot()`.

### Returns

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

See also:

**plotly.express.line** Plot y versus x as lines and/or markers (plotly).

**matplotlib.pyplot.plot** Plot y versus x as lines and/or markers (matplotlib).

## Examples

Basic plot.

For Series:

```
>>> s = ks.Series([1, 3, 2])
>>> s.plot.line()
```

For DataFrame:

The following example shows the populations for some animals over the years.

```
>>> df = ks.DataFrame({'pig': [20, 18, 489, 675, 1776],
...                    'horse': [4, 25, 281, 600, 1900]},
...                    index=[1990, 1997, 2003, 2009, 2014])
>>> df.plot.line()
```

The following example shows the relationship between both populations.

```
>>> df = ks.DataFrame({'pig': [20, 18, 489, 675, 1776],
...                    'horse': [4, 25, 281, 600, 1900]},
...                    index=[1990, 1997, 2003, 2009, 2014])
>>> df.plot.line(x='pig', y='horse')
```

### databricks.koalas.Series.plot.pie

`plot.pie(**kws)`

Generate a pie plot.

A pie plot is a proportional representation of the numerical data in a column. This function wraps `plotly.express.pie()` for the specified column.

#### Parameters

**y** [int or label, optional] Label or position of the column to plot. If not provided, `subplots=True` argument must be passed (matplotlib-only).

**\*\*kws** Keyword arguments to pass on to `Koalas.Series.plot()`.

#### Returns

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

### Examples

For Series:

```
>>> df = ks.DataFrame({'mass': [0.330, 4.87, 5.97],
...                    'radius': [2439.7, 6051.8, 6378.1]},
...                    index=['Mercury', 'Venus', 'Earth'])
>>> df.mass.plot.pie()
```

For DataFrame:

```
>>> df = ks.DataFrame({'mass': [0.330, 4.87, 5.97],
...                    'radius': [2439.7, 6051.8, 6378.1]},
...                    index=['Mercury', 'Venus', 'Earth'])
>>> df.plot.pie(y='mass')
```

**databricks.koalas.Series.plot.kde**

`plot.kde` (*bw\_method=None*, *ind=None*, *\*\*kwargs*)  
Generate Kernel Density Estimate plot using Gaussian kernels.

**Parameters**

**bw\_method** [scalar] The method used to calculate the estimator bandwidth. See `KernelDensity` in PySpark for more information.

**ind** [NumPy array or integer, optional] Evaluation points for the estimated PDF. If `None` (default), 1000 equally spaced points are used. If *ind* is a NumPy array, the KDE is evaluated at the points passed. If *ind* is an integer, *ind* number of equally spaced points are used.

**\*\*kwargs** [optional] Keyword arguments to pass on to `Koalas.Series.plot()`.

**Returns**

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

**Examples**

A scalar bandwidth should be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```
>>> s = ks.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> s.plot.kde(bw_method=0.3)
```

```
>>> s = ks.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> s.plot.kde(bw_method=3)
```

The *ind* parameter determines the evaluation points for the plot of the estimated KDF:

```
>>> s = ks.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> s.plot.kde(ind=[1, 2, 3, 4, 5], bw_method=0.3)
```

For `DataFrame`, it works in the same way as `Series`:

```
>>> df = ks.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> df.plot.kde(bw_method=0.3)
```

```
>>> df = ks.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> df.plot.kde(bw_method=3)
```

```
>>> df = ks.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> df.plot.kde(ind=[1, 2, 3, 4, 5, 6], bw_method=0.3)
```

**databricks.koalas.Series.hist**`Series.hist (bins=10, **kws)`

Draw one histogram of the DataFrame's columns. A [histogram](#) is a representation of the distribution of data. This function calls `plotting.backend.plot()`, on each series in the DataFrame, resulting in one histogram per column.

**Parameters**

**bins** [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

**\*\*kws** All other plotting keyword arguments to be passed to plotting backend.

**Returns**

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

**Examples**

Basic plot.

For Series:

```
>>> s = ks.Series([1, 3, 2])
>>> s.plot.hist()
```

For DataFrame:

```
>>> df = pd.DataFrame(
...     np.random.randint(1, 7, 6000),
...     columns=['one'])
>>> df['two'] = df['one'] + np.random.randint(1, 7, 6000)
>>> df = ks.from_pandas(df)
>>> df.plot.hist(bins=12, alpha=0.5)
```

**3.3.19 Serialization / IO / Conversion**

<code>Series.to_pandas()</code>	Return a pandas Series.
<code>Series.to_numpy()</code>	A NumPy ndarray representing the values in this DataFrame or Series.
<code>Series.to_list()</code>	Return a list of the values.
<code>Series.to_string([buf, na_rep, ...])</code>	Render a string representation of the Series.
<code>Series.to_dict([into])</code>	Convert Series to {label -> value} dict or dict-like object.
<code>Series.to_clipboard([excel, sep])</code>	Copy object to the system clipboard.
<code>Series.to_latex([buf, columns, col_space, ...])</code>	Render an object to a LaTeX tabular environment table.
<code>Series.to_markdown([buf, mode])</code>	Print Series or DataFrame in Markdown-friendly format.
<code>Series.to_json([path, compression, ...])</code>	Convert the object to a JSON string.
<code>Series.to_csv([path, sep, na_rep, columns, ...])</code>	Write object to a comma-separated values (csv) file.

continues on next page

Table 37 – continued from previous page

<code>Series.to_excel(excel_writer[, sheet_name, ...])</code>	Write object to an Excel sheet.
<code>Series.to_frame([name])</code>	Convert Series to DataFrame.

**databricks.koalas.Series.to\_pandas**`Series.to_pandas()` → `pandas.core.series.Series`

Return a pandas Series.

**Note:** This method should only be used if the resulting pandas object is expected to be small, as all the data is loaded into the driver's memory.

**Examples**

```
>>> df = ks.DataFrame([(0.2, .3), (.0, .6), (.6, .0), (.2, .1)], columns=['dogs',
→ 'cats'])
>>> df['dogs'].to_pandas()
0    0.2
1    0.0
2    0.6
3    0.2
Name: dogs, dtype: float64
```

**databricks.koalas.Series.to\_numpy**`Series.to_numpy()` → `numpy.ndarray`

A NumPy ndarray representing the values in this DataFrame or Series.

**Note:** This method should only be used if the resulting NumPy ndarray is expected to be small, as all the data is loaded into the driver's memory.

**Returns****numpy.ndarray****Examples**

```
>>> ks.DataFrame({"A": [1, 2], "B": [3, 4]}).to_numpy()
array([[1, 3],
       [2, 4]])
```

With heterogeneous data, the lowest common type will have to be used.

```
>>> ks.DataFrame({"A": [1, 2], "B": [3.0, 4.5]}).to_numpy()
array([[1. , 3. ],
       [2. , 4.5]])
```

For a mix of numeric and non-numeric types, the output array will have object dtype.



```
>>> df = ks.DataFrame({"A": [1, 2], "B": [3.0, 4.5], "C": pd.date_range('2000',
↳ periods=2)})
>>> df.to_numpy()
array([[1, 3.0, Timestamp('2000-01-01 00:00:00')],
       [2, 4.5, Timestamp('2000-01-02 00:00:00')]], dtype=object)
```

For Series,

```
>>> ks.Series(['a', 'b', 'a']).to_numpy()
array(['a', 'b', 'a'], dtype=object)
```

### **databricks.koalas.Series.to\_list**

**Series.to\_list()** → List

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

---

**Note:** This method should only be used if the resulting list is expected to be small, as all the data is loaded into the driver's memory.

---

### **databricks.koalas.Series.to\_string**

**Series.to\_string**(buf=None, na\_rep='NaN', float\_format=None, header=True, index=True, length=False, dtype=False, name=False, max\_rows=None) → str

Render a string representation of the Series.

---

**Note:** This method should only be used if the resulting pandas object is expected to be small, as all the data is loaded into the driver's memory. If the input is large, set max\_rows parameter.

---

#### **Parameters**

**buf** [StringIO-like, optional] buffer to write to

**na\_rep** [string, optional] string representation of NaN to use, default 'NaN'

**float\_format** [one-parameter function, optional] formatter function to apply to columns' elements if they are floats default None

**header** [boolean, default True] Add the Series header (index name)

**index** [bool, optional] Add index (row) labels, default True

**length** [boolean, default False] Add the Series length

**dtype** [boolean, default False] Add the Series dtype

**name** [boolean, default False] Add the Series name if not None

**max\_rows** [int, optional] Maximum number of rows to show before truncating. If None, show all.

#### **Returns**

**formatted** [string (if not buffer passed)]

### Examples

```
>>> df = ks.DataFrame([(0.2, .3), (.0, .6), (.6, .0), (.2, .1)], columns=['dogs',  
→ 'cats'])  
>>> print(df['dogs'].to_string())  
0    0.2  
1    0.0  
2    0.6  
3    0.2
```

```
>>> print(df['dogs'].to_string(max_rows=2))  
0    0.2  
1    0.0
```

### `databricks.koalas.Series.to_dict`

`Series.to_dict` (*into*=<class 'dict'>) → `collections.abc.Mapping`  
Convert Series to {label -> value} dict or dict-like object.

---

**Note:** This method should only be used if the resulting pandas DataFrame is expected to be small, as all the data is loaded into the driver's memory.

---

#### Parameters

**into** [class, default dict] The `collections.abc.Mapping` subclass to use as the return object. Can be the actual class or an empty instance of the mapping type you want. If you want a `collections.defaultdict`, you must pass it initialized.

#### Returns

**`collections.abc.Mapping`** Key-value representation of Series.

### Examples

```
>>> s = ks.Series([1, 2, 3, 4])  
>>> s_dict = s.to_dict()  
>>> sorted(s_dict.items())  
[(0, 1), (1, 2), (2, 3), (3, 4)]
```

```
>>> from collections import OrderedDict, defaultdict  
>>> s.to_dict(OrderedDict)  
OrderedDict([(0, 1), (1, 2), (2, 3), (3, 4)])
```

```
>>> dd = defaultdict(list)  
>>> s.to_dict(dd)  
defaultdict(<class 'list'>, {...})
```

**databricks.koalas.Series.to\_clipboard**

`Series.to_clipboard(excel=True, sep=None, **kwargs) → None`

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

---

**Note:** This method should only be used if the resulting DataFrame is expected to be small, as all the data is loaded into the driver's memory.

---

**Parameters**

**excel** [bool, default True]

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

**sep** [str, default '\t'] Field delimiter.

**\*\*kwargs** These parameters will be passed to `DataFrame.to_csv`.

See also:

[`read\_clipboard`](#) Read text from clipboard.

**Notes**

Requirements for your platform.

- Linux : *xclip*, or *xsel* (with *gtk* or *PyQt4* modules)
- Windows : none
- OS X : none

**Examples**

Copy the contents of a DataFrame to the clipboard.

```
>>> df = ks.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

This function also works for Series:

```
>>> df = ks.Series([1, 2, 3, 4, 5, 6, 7], name='x')
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # 0, 1
... # 1, 2
... # 2, 3
... # 3, 4
... # 4, 5
... # 5, 6
... # 6, 7
```

### **databricks.koalas.Series.to\_latex**

`Series.to_latex` (*buf=None, columns=None, col\_space=None, header=True, index=True, na\_rep='NaN', formatters=None, float\_format=None, sparsify=None, index\_names=True, bold\_rows=False, column\_format=None, longtable=None, escape=None, encoding=None, decimal='.', multicolumn=None, multirow=None*) → Optional[str]

Render an object to a LaTeX tabular environment table.

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires `usepackage{booktabs}`.

---

**Note:** This method should only be used if the resulting pandas object is expected to be small, as all the data is loaded into the driver's memory. If the input is large, consider alternative formats.

---

#### **Parameters**

**buf** [file descriptor or None] Buffer to write to. If None, the output is returned as a string.

**columns** [list of label, optional] The subset of columns to write. Writes all columns by default.

**col\_space** [int, optional] The minimum width of each column.

**header** [bool or list of str, default True] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.

**index** [bool, default True] Write row names (index).

**na\_rep** [str, default 'NaN'] Missing data representation.

**formatters** [list of functions or dict of {str: function}, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

**float\_format** [str, optional] Format string for floating point numbers.

**sparsify** [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row. By default, the value will be read from the config module.

**index\_names** [bool, default True] Prints the names of the indexes.

**bold\_rows** [bool, default False] Make the row labels bold in the output.

**column\_format** [str, optional] The columns format as specified in LaTeX table format e.g. 'rcl' for 3 columns. By default, 'l' will be used for all columns except columns of numbers, which default to 'r'.

**longtable** [bool, optional] By default, the value will be read from the pandas config module. Use a longtable environment instead of tabular. Requires adding a `usepackage{longtable}` to your LaTeX preamble.

**escape** [bool, optional] By default, the value will be read from the pandas config module. When set to False prevents from escaping latex special characters in column names.

**encoding** [str, optional] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

**decimal** [str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

**multicolumn** [bool, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

**multicolumn\_format** [str, default 'l'] The alignment for multicolumns, similar to column\_format The default will be read from the config module.

**multirow** [bool, default False] Use multirow to enhance MultiIndex rows. Requires adding a `usepackage{multirow}` to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

### Returns

**str or None** If buf is None, returns the resulting LaTeX format as a string. Otherwise returns None.

### See also:

**DataFrame.to\_string** Render a DataFrame to a console-friendly tabular output.

**DataFrame.to\_html** Render a DataFrame as an HTML table.

### Examples

```
>>> df = ks.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']},
...                    columns=['name', 'mask', 'weapon'])
>>> print(df.to_latex(index=False))
\begin{tabular}{lll}
\toprule
    name &    mask &    weapon \\
\midrule
    Raphael &    red &    sai \\
    Donatello & purple & bo staff \\
\bottomrule
\end{tabular}
```

**databricks.koalas.Series.to\_markdown**

`Series.to_markdown (buf=None, mode=None) → str`  
Print Series or DataFrame in Markdown-friendly format.

---

**Note:** This method should only be used if the resulting pandas object is expected to be small, as all the data is loaded into the driver's memory.

---

**Parameters**

**buf** [writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

**mode** [str, optional] Mode in which file is opened.

**\*\*kwargs** These parameters will be passed to *tabulate*.

**Returns**

**str** Series or DataFrame in Markdown-friendly format.

**Notes**

Requires the *tabulate* package.

**Examples**

```
>>> kser = ks.Series(["elk", "pig", "dog", "quetzal"], name="animal")
>>> print(kser.to_markdown())
|   | animal |
|---:|:-----|
| 0 | elk      |
| 1 | pig      |
| 2 | dog      |
| 3 | quetzal  |
```

```
>>> kdf = ks.DataFrame(
...     data={"animal_1": ["elk", "pig"], "animal_2": ["dog", "quetzal"]}
... )
>>> print(kdf.to_markdown())
|   | animal_1 | animal_2 |
|---:|:-----|:-----|
| 0 | elk      | dog      |
| 1 | pig      | quetzal  |
```

**databricks.koalas.Series.to\_json**

`Series.to_json` (*path=None, compression='uncompressed', num\_files=None, mode: str = 'overwrite', orient='records', lines=True, partition\_cols: Union[str, List[str], None] = None, index\_col: Union[str, List[str], None] = None, \*\*options*) → Optional[str]

Convert the object to a JSON string.

---

**Note:** Koalas `to_json` writes files to a path or URI. Unlike pandas', Koalas respects HDFS's property such as `'fs.default.name'`.

---



---

**Note:** Koalas writes JSON files into the directory, *path*, and writes multiple *part-...* files in the directory when *path* is specified. This behaviour was inherited from Apache Spark. The number of files can be controlled by *num\_files*.

---



---

**Note:** output JSON format is different from pandas'. It always use *orient='records'* for its output. This behaviour might have to change in the near future.

---

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

**Parameters**

**path** [string, optional] File path. If not specified, the result is returned as a string.

**lines** [bool, default True] If 'orient' is 'records' write out line delimited json format. Will throw ValueError if incorrect 'orient' since others are not list like. It should be always True for now.

**orient** [str, default 'records'] It should be always 'records' for now.

**compression** [{ 'gzip', 'bz2', 'xz', None}] A string representing the compression to use in the output file, only used when the first argument is a filename. By default, the compression is inferred from the filename.

**num\_files** [the number of files to be written in *path* directory when] this is a path.

**mode** [str { 'append', 'overwrite', 'ignore', 'error', 'errorifexists' },] default 'overwrite'. Specifies the behavior of the save operation when the destination exists already.

- 'append': Append the new data to existing data.
- 'overwrite': Overwrite existing data.
- 'ignore': Silently ignore this operation if data already exists.
- 'error' or 'errorifexists': Throw an exception if data already exists.

**partition\_cols** [str or list of str, optional, default None] Names of partitioning columns

**index\_col: str or list of str, optional, default: None** Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**options: keyword arguments for additional options specific to PySpark.** It is specific to PySpark's JSON options to pass. Check the options in PySpark's API documentation for `spark.write.json(...)`. It has a higher priority and overwrites all other options. This parameter only works when *path* is specified.

**Returns**

**str or None**

### Examples

```
>>> df = ks.DataFrame([[ 'a', 'b'], [ 'c', 'd']],
...                    columns=[ 'col 1', 'col 2'])
>>> df.to_json()
' [{"col 1": "a", "col 2": "b"}, {"col 1": "c", "col 2": "d"} ] '
```

```
>>> df['col 1'].to_json()
' [{"col 1": "a"}, {"col 1": "c"} ] '
```

```
>>> df.to_json(path=r'%s/to_json/foo.json' % path, num_files=1)
>>> ks.read_json(
...     path=r'%s/to_json/foo.json' % path
... ).sort_values(by="col 1")
   col 1 col 2
0      a     b
1      c     d
```

```
>>> df['col 1'].to_json(path=r'%s/to_json/foo.json' % path, num_files=1, index_
↳ col="index")
>>> ks.read_json(
...     path=r'%s/to_json/foo.json' % path, index_col="index"
... ).sort_values(by="col 1")
   col 1
index
0      a
1      c
```

### databricks.koalas.Series.to\_csv

`Series.to_csv(path=None, sep=',', na_rep="", columns=None, header=True, quotechar="'", date_format=None, escapechar=None, num_files=None, mode: str = 'overwrite', partition_cols: Union[str, List[str], None] = None, index_col: Union[str, List[str], None] = None, **options) → Optional[str]`

Write object to a comma-separated values (csv) file.

---

**Note:** Koalas `to_csv` writes files to a path or URI. Unlike pandas', Koalas respects HDFS's property such as `'fs.default.name'`.

---

---

**Note:** Koalas writes CSV files into the directory, `path`, and writes multiple `part-...` files in the directory when `path` is specified. This behaviour was inherited from Apache Spark. The number of files can be controlled by `num_files`.

---

### Parameters

**path** [str, default None] File path. If None is provided the result is returned as a string.

**sep** [str, default ','] String of length 1. Field delimiter for the output file.



**na\_rep** [str, default ''] Missing data representation.

**columns** [sequence, optional] Columns to write.

**header** [bool or list of str, default True] Write out the column names. If a list of strings is given it is assumed to be aliases for the column names.

**quotechar** [str, default '"'] String of length 1. Character used to quote fields.

**date\_format** [str, default None] Format string for datetime objects.

**escapechar** [str, default None] String of length 1. Character used to escape *sep* and *quotechar* when appropriate.

**num\_files** [the number of files to be written in *path* directory when] this is a path.

**mode** [str { 'append', 'overwrite', 'ignore', 'error', 'errorifexists' },] default 'overwrite'. Specifies the behavior of the save operation when the destination exists already.

- 'append': Append the new data to existing data.
- 'overwrite': Overwrite existing data.
- 'ignore': Silently ignore this operation if data already exists.
- 'error' or 'errorifexists': Throw an exception if data already exists.

**partition\_cols** [str or list of str, optional, default None] Names of partitioning columns

**index\_col: str or list of str, optional, default: None** Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**options: keyword arguments for additional options specific to PySpark.** This kwargs are specific to PySpark's CSV options to pass. Check the options in PySpark's API documentation for `spark.write.csv(...)`. It has higher priority and overwrites all other options. This parameter only works when *path* is specified.

## Returns

str or None

See also:

[`read\_csv`](#)

[`DataFrame.to\_delta`](#)

[`DataFrame.to\_table`](#)

[`DataFrame.to\_parquet`](#)

[`DataFrame.to\_spark\_io`](#)

## Examples

```
>>> df = ks.DataFrame(dict(
...     date=list(pd.date_range('2012-1-1 12:00:00', periods=3, freq='M')),
...     country=['KR', 'US', 'JP'],
...     code=[1, 2, 3]), columns=['date', 'country', 'code'])
>>> df.sort_values(by="date")
           date country  code
... 2012-01-31 12:00:00    KR    1
... 2012-02-29 12:00:00    US    2
... 2012-03-31 12:00:00    JP    3
```

```
>>> print(df.to_csv())
date,country,code
2012-01-31 12:00:00,KR,1
2012-02-29 12:00:00,US,2
2012-03-31 12:00:00,JP,3
```

```
>>> df.cummax().to_csv(path=r'%s/to_csv/foo.csv' % path, num_files=1)
>>> ks.read_csv(
...     path=r'%s/to_csv/foo.csv' % path
... ).sort_values(by="date")
           date country  code
... 2012-01-31 12:00:00    KR    1
... 2012-02-29 12:00:00    US    2
... 2012-03-31 12:00:00    US    3
```

In case of Series,

```
>>> print(df.date.to_csv())
date
2012-01-31 12:00:00
2012-02-29 12:00:00
2012-03-31 12:00:00
```

```
>>> df.date.to_csv(path=r'%s/to_csv/foo.csv' % path, num_files=1)
>>> ks.read_csv(
...     path=r'%s/to_csv/foo.csv' % path
... ).sort_values(by="date")
           date
... 2012-01-31 12:00:00
... 2012-02-29 12:00:00
... 2012-03-31 12:00:00
```

You can preserve the index in the roundtrip as below.

```
>>> df.set_index("country", append=True, inplace=True)
>>> df.date.to_csv(
...     path=r'%s/to_csv/bar.csv' % path,
...     num_files=1,
...     index_col=["index1", "index2"])
>>> ks.read_csv(
...     path=r'%s/to_csv/bar.csv' % path, index_col=["index1", "index2"]
... ).sort_values(by="date")
           date
index1 index2
```

(continues on next page)

(continued from previous page)

```

...      ...      2012-01-31 12:00:00
...      ...      2012-02-29 12:00:00
...      ...      2012-03-31 12:00:00

```

**databricks.koalas.Series.to\_excel**

`Series.to_excel` (*excel\_writer*, *sheet\_name*='Sheet1', *na\_rep*="", *float\_format*=None, *columns*=None, *header*=True, *index*=True, *index\_label*=None, *startrow*=0, *startcol*=0, *engine*=None, *merge\_cells*=True, *encoding*=None, *inf\_rep*='inf', *verbose*=True, *freeze\_panes*=None)  
 → None  
 Write object to an Excel sheet.

---

**Note:** This method should only be used if the resulting DataFrame is expected to be small, as all the data is loaded into the driver's memory.

---

To write a single object to an Excel .xlsx file it is only necessary to specify a target file name. To write to multiple sheets it is necessary to create an *ExcelWriter* object with a target file name, and specify a sheet in the file to write to.

Multiple sheets may be written to by specifying unique *sheet\_name*. With all data written to the file it is necessary to save the changes. Note that creating an *ExcelWriter* object with a file name that already exists will result in the contents of the existing file being erased.

**Parameters**

**excel\_writer** [str or ExcelWriter object] File path or existing ExcelWriter.

**sheet\_name** [str, default 'Sheet1'] Name of sheet which will contain DataFrame.

**na\_rep** [str, default ''] Missing data representation.

**float\_format** [str, optional] Format string for floating point numbers. For example `float_format="%%.2f"` will format 0.1234 to 0.12.

**columns** [sequence or list of str, optional] Columns to write.

**header** [bool or list of str, default True] Write out the column names. If a list of string is given it is assumed to be aliases for the column names.

**index** [bool, default True] Write row names (index).

**index\_label** [str or sequence, optional] Column label for index column(s) if desired. If not specified, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

**startrow** [int, default 0] Upper left cell row to dump data frame.

**startcol** [int, default 0] Upper left cell column to dump data frame.

**engine** [str, optional] Write engine to use, 'openpyxl' or 'xlsxwriter'. You can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

**merge\_cells** [bool, default True] Write MultiIndex and Hierarchical Rows as merged cells.

**encoding** [str, optional] Encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

**inf\_rep** [str, default 'inf'] Representation for infinity (there is no native representation for infinity in Excel).

**verbose** [bool, default True] Display more information in the error logs.

**freeze\_panes** [tuple of int (length 2), optional] Specifies the one-based bottommost row and rightmost column that is to be frozen.

See also:

[`read\_excel`](#) Read Excel file.

## Notes

Once a workbook has been saved it is not possible write further data without rewriting the whole workbook.

## Examples

Create, write to and save a workbook:

```
>>> df1 = ks.DataFrame([[ 'a', 'b'], [ 'c', 'd']],
...                     index=[ 'row 1', 'row 2'],
...                     columns=[ 'col 1', 'col 2'])
>>> df1.to_excel("output.xlsx")
```

To specify the sheet name:

```
>>> df1.to_excel("output.xlsx")
>>> df1.to_excel("output.xlsx",
...             sheet_name='Sheet_name_1')
```

If you wish to write to more than one sheet in the workbook, it is necessary to specify an ExcelWriter object:

```
>>> with pd.ExcelWriter('output.xlsx') as writer:
...     df1.to_excel(writer, sheet_name='Sheet_name_1')
...     df2.to_excel(writer, sheet_name='Sheet_name_2')
```

To set the library that is used to write the Excel file, you can pass the *engine* keyword (the default engine is automatically chosen depending on the file extension):

```
>>> df1.to_excel('output1.xlsx', engine='xlsxwriter')
```

## `databricks.koalas.Series.to_frame`

`Series.to_frame(name: Union[Any, Tuple] = None) → databricks.koalas.frame.DataFrame`  
Convert Series to DataFrame.

### Parameters

**name** [object, default None] The passed name should substitute for the series name (if it has one).

### Returns

**DataFrame** DataFrame representation of Series.

## Examples

```
>>> s = ks.Series(["a", "b", "c"])
>>> s.to_frame()
   0
0  a
1  b
2  c
```

```
>>> s = ks.Series(["a", "b", "c"], name="vals")
>>> s.to_frame()
   vals
0     a
1     b
2     c
```

### 3.3.20 Koalas-specific

`Series.koalas` provides Koalas-specific features that exists only in Koalas. These can be accessed by `Series.koalas.<function/property>`.

---

<code>Series.koalas.transform_batch(func, *args, ...)</code>	Transform the data with the function that takes pandas Series and outputs pandas Series.
--	--

---

#### `databricks.koalas.Series.koalas.transform_batch`

`koalas.transform_batch(func, *args, **kwargs) → Series`

Transform the data with the function that takes pandas Series and outputs pandas Series. The pandas Series given to the function is of a batch used internally.

See also [Transform and apply a function](#).

---

**Note:** the *func* is unable to access to the whole input series. Koalas internally splits the input series into multiple batches and calls *func* with each batch multiple times. Therefore, operations such as global aggregations are impossible. See the example below.

```
>>> # This case does not return the length of whole frame but of the batch_
↳internally
... # used.
... def length(pser) -> ks.Series[int]:
...     return pd.Series([len(pser)] * len(pser))
...
>>> df = ks.DataFrame({'A': range(1000)})
>>> df.A.koalas.transform_batch(length)
   c0
0   83
1   83
2   83
...
```

---

**Note:** this API executes the function once to infer the type which is potentially expensive, for instance, when

the dataset is created after aggregations or sorting.

To avoid this, specify return type in `func`, for instance, as below:

```
>>> def plus_one(x) -> ks.Series[int]:  
...     return x + 1
```

### Parameters

**func** [function] Function to apply to each pandas frame.

**\*args** Positional arguments to pass to func.

**\*\*kwargs** Keyword arguments to pass to func.

### Returns

**DataFrame**

See also:

**DataFrame.koalas.apply\_batch** Similar but it takes pandas DataFrame as its internal batch.

### Examples

```
>>> df = ks.DataFrame([(1, 2), (3, 4), (5, 6)], columns=['A', 'B'])  
>>> df  
   A  B  
0  1  2  
1  3  4  
2  5  6
```

```
>>> def plus_one_func(pser) -> ks.Series[np.int64]:  
...     return pser + 1  
>>> df.A.koalas.transform_batch(plus_one_func)  
0    2  
1    4  
2    6  
Name: A, dtype: int64
```

You can also omit the type hints so Koalas infers the return schema as below:

```
>>> df.A.koalas.transform_batch(lambda pser: pser + 1)  
0    2  
1    4  
2    6  
Name: A, dtype: int64
```

You can also specify extra arguments.

```
>>> def plus_one_func(pser, a, b, c=3) -> ks.Series[np.int64]:  
...     return pser + a + b + c  
>>> df.A.koalas.transform_batch(plus_one_func, 1, b=2)  
0    7  
1    9  
2   11  
Name: A, dtype: int64
```

You can also use `np.ufunc` and built-in functions as input.

```
>>> df.A.koalas.transform_batch(np.add, 10)
0      11
1      13
2      15
Name: A, dtype: int64
```

```
>>> (df * -1).A.koalas.transform_batch(abs)
0      1
1      3
2      5
Name: A, dtype: int64
```

## 3.4 DataFrame

### 3.4.1 Constructor

---

<code>DataFrame([data, index, columns, dtype, copy])</code>	Koalas DataFrame that corresponds to pandas DataFrame logically.
---	--

---

#### `databricks.koalas.DataFrame`

**class** `databricks.koalas.DataFrame` (*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=False*)

Koalas DataFrame that corresponds to pandas DataFrame logically. This holds Spark DataFrame internally.

**Variables** `_internal` – an internal immutable Frame to manage metadata.

#### Parameters

**data** [numpy ndarray (structured or homogeneous), dict, pandas DataFrame, Spark DataFrame or Koalas Series] Dict can contain Series, arrays, constants, or list-like objects. If data is a dict, argument order is maintained for Python 3.6 and later. Note that if *data* is a pandas DataFrame, a Spark DataFrame, and a Koalas Series, other arguments should not be used.

**index** [Index or array-like] Index to use for resulting frame. Will default to RangeIndex if no indexing information part of input data and no index provided

**columns** [Index or array-like] Column labels to use for resulting frame. Will default to RangeIndex (0, 1, 2, ..., n) if no column labels are provided

**dtype** [dtype, default None] Data type to force. Only a single dtype is allowed. If None, infer

**copy** [boolean, default False] Copy data from inputs. Only affects DataFrame / 2d ndarray input

## Examples

Constructing DataFrame from a dictionary.

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = ks.DataFrame(data=d, columns=['col1', 'col2'])
>>> df
   col1  col2
0      1     3
1      2     4
```

Constructing DataFrame from pandas DataFrame

```
>>> df = ks.DataFrame(pd.DataFrame(data=d, columns=['col1', 'col2']))
>>> df
   col1  col2
0      1     3
1      2     4
```

Notice that the inferred dtype is int64.

```
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

To enforce a single dtype:

```
>>> df = ks.DataFrame(data=d, dtype=np.int8)
>>> df.dtypes
col1    int8
col2    int8
dtype: object
```

Constructing DataFrame from numpy ndarray:

```
>>> df2 = ks.DataFrame(np.random.randint(low=0, high=10, size=(5, 5)),
...                     columns=['a', 'b', 'c', 'd', 'e'])
>>> df2
   a  b  c  d  e
0  3  1  4  9  8
1  4  8  4  8  4
2  7  6  5  6  7
3  8  7  9  1  0
4  2  5  4  3  9
```

\_\_init\_\_ (data=None, index=None, columns=None, dtype=None, copy=False)  
Initialize self. See help(type(self)) for accurate signature.



## Methods

<code>__init__([data, index, columns, dtype, copy])</code>	Initialize self.
<code>abs()</code>	Return a Series/DataFrame with absolute numeric value of each element.
<code>add(other)</code>	Get Addition of dataframe and other, element-wise (binary operator +).
<code>add_prefix(prefix)</code>	Prefix labels with string <i>prefix</i> .
<code>add_suffix(suffix)</code>	Suffix labels with string <i>suffix</i> .
<code>agg(func)</code>	Aggregate using one or more operations over the specified axis.
<code>aggregate(func)</code>	Aggregate using one or more operations over the specified axis.
<code>align(other[, join, axis, copy])</code>	Align two objects on their axes with the specified join method.
<code>all([axis])</code>	Return whether all elements are True.
<code>any([axis])</code>	Return whether any element is True.
<code>append(other[, ignore_index, ...])</code>	Append rows of other to the end of caller, returning a new object.
<code>apply(func[, axis, args])</code>	Apply a function along an axis of the DataFrame.
<code>apply_batch(func[, args])</code>	Apply a function that takes pandas DataFrame and outputs pandas DataFrame.
<code>applymap(func)</code>	Apply a function to a Dataframe elementwise.
<code>assign(**kwargs)</code>	Assign new columns to a DataFrame.
<code>astype(dtype)</code>	Cast a Koalas object to a specified dtype <i>dtype</i> .
<code>at_time(time[, asof, axis])</code>	Select values at particular time of day (e.g., 9:30AM).
<code>backfill([axis, inplace, limit])</code>	Synonym for <i>DataFrame.fillna()</i> or <i>Series.fillna()</i> with <i>method='bfill'</i> .
<code>between_time(start_time, end_time[, ...])</code>	Select values between particular times of the day (e.g., 9:00-9:30 AM).
<code>bfill([axis, inplace, limit])</code>	Synonym for <i>DataFrame.fillna()</i> or <i>Series.fillna()</i> with <i>method='bfill'</i> .
<code>bool()</code>	Return the bool of a single element in the current object.
<code>cache()</code>	Yields and caches the current DataFrame.
<code>clip([lower, upper])</code>	Trim values at input threshold(s).
<code>copy([deep])</code>	Make a copy of this object's indices and data.
<code>corr([method])</code>	Compute pairwise correlation of columns, excluding NA/null values.
<code>count([axis, numeric_only])</code>	Count non-NA cells for each column.
<code>cummax([skipna])</code>	Return cumulative maximum over a DataFrame or Series axis.
<code>cummin([skipna])</code>	Return cumulative minimum over a DataFrame or Series axis.
<code>cumprod([skipna])</code>	Return cumulative product over a DataFrame or Series axis.
<code>cumsum([skipna])</code>	Return cumulative sum over a DataFrame or Series axis.

continues on next page

Table 40 – continued from previous page

<code>describe([percentiles])</code>	Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
<code>diff([periods, axis])</code>	First discrete difference of element.
<code>div(other)</code>	Get Floating division of dataframe and other, element-wise (binary operator /).
<code>divide(other)</code>	Get Floating division of dataframe and other, element-wise (binary operator /).
<code>dot(other)</code>	Compute the matrix multiplication between the DataFrame and other.
<code>drop([labels, axis, columns])</code>	Drop specified labels from columns.
<code>drop_duplicates([subset, keep, inplace])</code>	Return DataFrame with duplicate rows removed, optionally only considering certain columns.
<code>droplevel(level[, axis])</code>	Return DataFrame with requested index / column level(s) removed.
<code>dropna([axis, how, thresh, subset, inplace])</code>	Remove missing values.
<code>duplicated([subset, keep])</code>	Return boolean Series denoting duplicate rows, optionally only considering certain columns.
<code>eq(other)</code>	Compare if the current value is equal to the other.
<code>equals(other)</code>	Compare if the current value is equal to the other.
<code>eval(expr[, inplace])</code>	Evaluate a string describing operations on DataFrame columns.
<code>expanding([min_periods])</code>	Provide expanding transformations.
<code>explain([extended, mode])</code>	Prints the underlying (logical and physical) Spark plans to the console for debugging purpose.
<code>explode(column)</code>	Transform each element of a list-like to a row, replicating index values.
<code>ffill([axis, inplace, limit])</code>	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='ffill'</code> .
<code>fillna([value, method, axis, inplace, limit])</code>	Fill NA/NaN values.
<code>filter([items, like, regex, axis])</code>	Subset rows or columns of dataframe according to labels in the specified index.
<code>first(offset)</code>	Select first periods of time series data based on a date offset.
<code>first_valid_index()</code>	Retrieves the index of the first valid value.
<code>floordiv(other)</code>	Get Integer division of dataframe and other, element-wise (binary operator //).
<code>from_dict(data[, orient, dtype, columns])</code>	Construct DataFrame from dict of array-like or dicts.
<code>from_records(data[, index, exclude, ...])</code>	Convert structured or record ndarray to DataFrame.
<code>ge(other)</code>	Compare if the current value is greater than or equal to the other.
<code>get(key[, default])</code>	Get item from object for given key (DataFrame column, Panel slice, etc.).
<code>get_dtype_counts()</code>	Return counts of unique dtypes in this object.
<code>groupby(by[, axis, as_index, dropna])</code>	Group DataFrame or Series using a Series of columns.
<code>gt(other)</code>	Compare if the current value is greater than the other.
<code>head([n])</code>	Return the first <i>n</i> rows.
<code>hint(name, *parameters)</code>	Specifies some hint on the current DataFrame.
<code>hist([bins])</code>	Draw one histogram of the DataFrame's columns.

continues on next page

Table 40 – continued from previous page

<code>idxmax([axis])</code>	Return index of first occurrence of maximum over requested axis.
<code>idxmin([axis])</code>	Return index of first occurrence of minimum over requested axis.
<code>info([verbose, buf, max_cols, null_counts])</code>	Print a concise summary of a DataFrame.
<code>insert(loc, column, value[, allow_duplicates])</code>	Insert column into DataFrame at specified location.
<code>isin(values)</code>	Whether each element in the DataFrame is contained in values.
<code>isna()</code>	Detects missing values for items in the current Dataframe.
<code>isnull()</code>	Detects missing values for items in the current Dataframe.
<code>items()</code>	This is an alias of <code>iteritems</code> .
<code>iteritems()</code>	Iterator over (column name, Series) pairs.
<code>iterrows()</code>	Iterate over DataFrame rows as (index, Series) pairs.
<code>itertuples([index, name])</code>	Iterate over DataFrame rows as namedtuples.
<code>join(right[, on, how, lsuffix, rsuffix])</code>	Join columns of another DataFrame.
<code>kde([bw_method, ind])</code>	Generate Kernel Density Estimate plot using Gaussian kernels.
<code>keys()</code>	Return alias for columns.
<code>kurt([axis, numeric_only])</code>	Return unbiased kurtosis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>kurtosis([axis, numeric_only])</code>	Return unbiased kurtosis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>last(offset)</code>	Select final periods of time series data based on a date offset.
<code>last_valid_index()</code>	Return index for last non-NA/null value.
<code>le(other)</code>	Compare if the current value is less than or equal to the other.
<code>lt(other)</code>	Compare if the current value is less than the other.
<code>mad([axis])</code>	Return the mean absolute deviation of values.
<code>map_in_pandas(func)</code>	Apply a function that takes pandas DataFrame and outputs pandas DataFrame.
<code>mask(cond[, other])</code>	Replace values where the condition is True.
<code>max([axis, numeric_only])</code>	Return the maximum of the values.
<code>mean([axis, numeric_only])</code>	Return the mean of the values.
<code>median([axis, numeric_only, accuracy])</code>	Return the median of the values for the requested axis.
<code>melt([id_vars, value_vars, var_name, value_name])</code>	Unpivot a DataFrame from wide format to long format, optionally leaving identifier variables set.
<code>merge(right[, how, on, left_on, right_on, ...])</code>	Merge DataFrame objects with a database-style join.
<code>min([axis, numeric_only])</code>	Return the minimum of the values.
<code>mod(other)</code>	Get Modulo of dataframe and other, element-wise (binary operator %).
<code>mul(other)</code>	Get Multiplication of dataframe and other, element-wise (binary operator *).
<code>multiply(other)</code>	Get Multiplication of dataframe and other, element-wise (binary operator *).
<code>ne(other)</code>	Compare if the current value is not equal to the other.
<code>nlargest(n, columns)</code>	Return the first <i>n</i> rows ordered by <i>columns</i> in descending order.

continues on next page

Table 40 – continued from previous page

<code>notna()</code>	Detects non-missing values for items in the current DataFrame.
<code>notnull()</code>	Detects non-missing values for items in the current DataFrame.
<code>nsmallest(n, columns)</code>	Return the first <i>n</i> rows ordered by <i>columns</i> in ascending order.
<code>nunique([axis, dropna, approx, rsd])</code>	Return number of unique elements in the object.
<code>pad([axis, inplace, limit])</code>	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='ffill'</code> .
<code>pct_change([periods])</code>	Percentage change between the current and a prior element.
<code>persist([storage_level])</code>	Yields and caches the current DataFrame with a specific <code>StorageLevel</code> .
<code>pipe(func, *args, **kwargs)</code>	Apply <code>func(self, *args, **kwargs)</code> .
<code>pivot([index, columns, values])</code>	Return reshaped DataFrame organized by given index / column values.
<code>pivot_table([values, index, columns, ...])</code>	Create a spreadsheet-style pivot table as a DataFrame.
<code>pop(item)</code>	Return item and drop from frame.
<code>pow(other)</code>	Get Exponential power of series of dataframe and other, element-wise (binary operator <code>**</code> ).
<code>print_schema([index_col])</code>	Prints out the underlying Spark schema in the tree format.
<code>prod([axis, numeric_only, min_count])</code>	Return the product of the values.
<code>product([axis, numeric_only, min_count])</code>	Return the product of the values.
<code>quantile([q, axis, numeric_only, accuracy])</code>	Return value at the given quantile.
<code>query(expr[, inplace])</code>	Query the columns of a DataFrame with a boolean expression.
<code>radd(other)</code>	Get Addition of dataframe and other, element-wise (binary operator <code>+</code> ).
<code>rank([method, ascending])</code>	Compute numerical data ranks (1 through <i>n</i> ) along axis.
<code>rdiv(other)</code>	Get Floating division of dataframe and other, element-wise (binary operator <code>/</code> ).
<code>reindex([labels, index, columns, axis, ...])</code>	Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
<code>reindex_like(other[, copy])</code>	Return a DataFrame with matching indices as other object.
<code>rename([mapper, index, columns, axis, ...])</code>	Alter axes labels.
<code>rename_axis([mapper, index, columns, axis, ...])</code>	Set the name of the axis for the index or columns.
<code>replace([to_replace, value, inplace, limit, ...])</code>	Returns a new DataFrame replacing a value with another value.
<code>reset_index([level, drop, inplace, ...])</code>	Reset the index, or a level of it.
<code>rfloordiv(other)</code>	Get Integer division of dataframe and other, element-wise (binary operator <code>//</code> ).
<code>rmod(other)</code>	Get Modulo of dataframe and other, element-wise (binary operator <code>%</code> ).
<code>rmul(other)</code>	Get Multiplication of dataframe and other, element-wise (binary operator <code>*</code> ).

continues on next page

Table 40 – continued from previous page

<code>rolling(window[, min_periods])</code>	Provide rolling transformations.
<code>round([decimals])</code>	Round a DataFrame to a variable number of decimal places.
<code>rpow(other)</code>	Get Exponential power of dataframe and other, element-wise (binary operator <code>**</code> ).
<code>rsub(other)</code>	Get Subtraction of dataframe and other, element-wise (binary operator <code>-</code> ).
<code>rtruediv(other)</code>	Get Floating division of dataframe and other, element-wise (binary operator <code>/</code> ).
<code>sample([n, frac, replace, random_state])</code>	Return a random sample of items from an axis of object.
<code>select_dtypes([include, exclude])</code>	Return a subset of the DataFrame's columns based on the column dtypes.
<code>sem([axis, ddof, numeric_only])</code>	Return unbiased standard error of the mean over requested axis.
<code>set_index(keys[, drop, append, inplace])</code>	Set the DataFrame index (row labels) using one or more existing columns.
<code>shift([periods, fill_value])</code>	Shift DataFrame by desired number of periods.
<code>skew([axis, numeric_only])</code>	Return unbiased skew normalized by N-1.
<code>sort_index([axis, level, ascending, ...])</code>	Sort object by labels (along an axis)
<code>sort_values(by[, ascending, inplace, ...])</code>	Sort by the values along either axis.
<code>spark_schema([index_col])</code>	Returns the underlying Spark schema.
<code>squeeze([axis])</code>	Squeeze 1 dimensional axis objects into scalars.
<code>stack()</code>	Stack the prescribed level(s) from columns to index.
<code>std([axis, ddof, numeric_only])</code>	Return sample standard deviation.
<code>sub(other)</code>	Get Subtraction of dataframe and other, element-wise (binary operator <code>-</code> ).
<code>subtract(other)</code>	Get Subtraction of dataframe and other, element-wise (binary operator <code>-</code> ).
<code>sum([axis, numeric_only, min_count])</code>	Return the sum of the values.
<code>swapaxes(i, j[, copy])</code>	Interchange axes and swap values axes appropriately.
<code>swaplevel([i, j, axis])</code>	Swap levels i and j in a MultiIndex on a particular axis.
<code>tail([n])</code>	Return the last <i>n</i> rows.
<code>take(indices[, axis])</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>toPandas()</code>	Return a pandas DataFrame.
<code>to_clipboard([excel, sep])</code>	Copy object to the system clipboard.
<code>to_csv([path, sep, na_rep, columns, header, ...])</code>	Write object to a comma-separated values (csv) file.
<code>to_delta(path[, mode, partition_cols, index_col])</code>	Write the DataFrame out as a Delta Lake table.
<code>to_dict([orient, into])</code>	Convert the DataFrame to a dictionary.
<code>to_excel(excel_writer[, sheet_name, na_rep, ...])</code>	Write object to an Excel sheet.
<code>to_html([buf, columns, col_space, header, ...])</code>	Render a DataFrame as an HTML table.
<code>to_json([path, compression, num_files, ...])</code>	Convert the object to a JSON string.
<code>to_koalas([index_col])</code>	Converts the existing DataFrame into a Koalas DataFrame.
<code>to_latex([buf, columns, col_space, header, ...])</code>	Render an object to a LaTeX tabular environment table.
<code>to_markdown([buf, mode])</code>	Print Series or DataFrame in Markdown-friendly format.

continues on next page

Table 40 – continued from previous page

<code>to_numpy()</code>	A NumPy ndarray representing the values in this DataFrame or Series.
<code>to_orc(path[, mode, partition_cols, index_col])</code>	Write the DataFrame out as a ORC file or directory.
<code>to_pandas()</code>	Return a pandas DataFrame.
<code>to_parquet(path[, mode, partition_cols, ...])</code>	Write the DataFrame out as a Parquet file or directory.
<code>to_records([index, column_dtypes, index_dtypes])</code>	Convert DataFrame to a NumPy record array.
<code>to_spark([index_col])</code>	Spark related features.
<code>to_spark_io([path, format, mode, ...])</code>	Write the DataFrame out to a Spark data source.
<code>to_string([buf, columns, col_space, header, ...])</code>	Render a DataFrame to a console-friendly tabular output.
<code>to_table(name[, format, mode, ...])</code>	Write the DataFrame into a Spark table.
<code>transform(func[, axis])</code>	Call <code>func</code> on self producing a Series with transformed values and that has the same length as its input.
<code>transform_batch(func, *args, **kwargs)</code>	Transform chunks with a function that takes pandas DataFrame and outputs pandas DataFrame.
<code>transpose()</code>	Transpose index and columns.
<code>truediv(other)</code>	Get Floating division of dataframe and other, element-wise (binary operator <code>/</code> ).
<code>truncate([before, after, axis, copy])</code>	Truncate a Series or DataFrame before and after some index value.
<code>unstack()</code>	Pivot the (necessarily hierarchical) index labels.
<code>update(other[, join, overwrite])</code>	Modify in place using non-NA values from another DataFrame.
<code>var([axis, ddof, numeric_only])</code>	Return unbiased variance.
<code>where(cond[, other])</code>	Replace values where the condition is False.
<code>xs(key[, axis, level])</code>	Return cross-section from the DataFrame.

### Attributes

<code>T</code>	Transpose index and columns.
<code>at</code>	Access a single value for a row/column label pair.
<code>axes</code>	Return a list representing the axes of the DataFrame.
<code>columns</code>	The column labels of the DataFrame.
<code>dtypes</code>	Return the dtypes in the DataFrame.
<code>empty</code>	Returns true if the current DataFrame is empty.
<code>iat</code>	Access a single value for a row/column pair by integer position.
<code>iloc</code>	Purely integer-location based indexing for selection by position.
<code>index</code>	The index (row labels) Column of the DataFrame.
<code>is_dataframe</code>	
<code>loc</code>	Access a group of rows and columns by label(s) or a boolean Series.
<code>ndim</code>	Return an int representing the number of array dimensions.

continues on next page

Table 41 – continued from previous page

<i>shape</i>	Return a tuple representing the dimensionality of the DataFrame.
<i>size</i>	Return an int representing the number of elements in this object.
<i>style</i>	Property returning a Styler object containing methods for building a styled HTML representation for the DataFrame.
<i>values</i>	Return a Numpy representation of the DataFrame or the Series.

### 3.4.2 Attributes and underlying data

<i>DataFrame.index</i>	The index (row labels) Column of the DataFrame.
<i>DataFrame.columns</i>	The column labels of the DataFrame.
<i>DataFrame.empty</i>	Returns true if the current DataFrame is empty.

#### **databricks.koalas.DataFrame.index**

**property** `DataFrame.index`

The index (row labels) Column of the DataFrame.

Currently not supported when the DataFrame has no index.

**See also:**

[\*Index\*](#)

#### **databricks.koalas.DataFrame.columns**

**property** `DataFrame.columns`

The column labels of the DataFrame.

#### **databricks.koalas.DataFrame.empty**

**property** `DataFrame.empty`

Returns true if the current DataFrame is empty. Otherwise, returns false.

#### **Examples**

```
>>> ks.range(10).empty
False
```

```
>>> ks.range(0).empty
True
```

```
>>> ks.DataFrame({}, index=list('abc')).empty
True
```

<code>DataFrame.dtypes</code>	Return the dtypes in the DataFrame.
<code>DataFrame.shape</code>	Return a tuple representing the dimensionality of the DataFrame.
<code>DataFrame.axes</code>	Return a list representing the axes of the DataFrame.
<code>DataFrame.ndim</code>	Return an int representing the number of array dimensions.
<code>DataFrame.size</code>	Return an int representing the number of elements in this object.
<code>DataFrame.select_dtypes([include, exclude])</code>	Return a subset of the DataFrame's columns based on the column dtypes.
<code>DataFrame.values</code>	Return a Numpy representation of the DataFrame or the Series.

## `databricks.koalas.DataFrame.dtypes`

### **property** `DataFrame.dtypes`

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the object dtype.

#### **Returns**

**pd.Series** The data type of each column.

### **Examples**

```
>>> df = ks.DataFrame({'a': list('abc'),
...                   'b': list(range(1, 4)),
...                   'c': np.arange(3, 6).astype('i1'),
...                   'd': np.arange(4.0, 7.0, dtype='float64'),
...                   'e': [True, False, True],
...                   'f': pd.date_range('20130101', periods=3)},
...                   columns=['a', 'b', 'c', 'd', 'e', 'f'])
>>> df.dtypes
a          object
b          int64
c           int8
d         float64
e           bool
f  datetime64[ns]
dtype: object
```



**databricks.koalas.DataFrame.shape****property** DataFrame.**shape**

Return a tuple representing the dimensionality of the DataFrame.

**Examples**

```
>>> df = ks.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)
```

```
>>> df = ks.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                    'col3': [5, 6]})
>>> df.shape
(2, 3)
```

**databricks.koalas.DataFrame.axes****property** DataFrame.**axes**

Return a list representing the axes of the DataFrame.

It has the row axis labels and column axis labels as the only members. They are returned in that order.

**Examples**

```
>>> df = ks.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[Int64Index([0, 1], dtype='int64'), Index(['col1', 'col2'], dtype='object')]
```

**databricks.koalas.DataFrame.ndim****property** DataFrame.**ndim**

Return an int representing the number of array dimensions.

return 2 for DataFrame.

**Examples**

```
>>> df = ks.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=['cobra', 'viper', None],
...                    columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
cobra         1      2
viper         4      5
NaN          7      8
>>> df.ndim
2
```

**databricks.koalas.DataFrame.size****property** DataFrame.**size**

Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

**Examples**

```
>>> s = ks.Series({'a': 1, 'b': 2, 'c': None})
>>> s.size
3
```

```
>>> df = ks.DataFrame({'col1': [1, 2, None], 'col2': [3, 4, None]})
>>> df.size
6
```

```
>>> df = ks.DataFrame(index=[1, 2, None])
>>> df.size
0
```

**databricks.koalas.DataFrame.select\_dtypes**

DataFrame.**select\_dtypes** (*include=None, exclude=None*) → databricks.koalas.frame.DataFrame

Return a subset of the DataFrame's columns based on the column dtypes.

**Parameters**

**include, exclude** [scalar or list-like] A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied. It also takes Spark SQL DDL type strings, for instance, 'string' and 'date'.

**Returns**

**DataFrame** The subset of the frame including the dtypes in *include* and excluding the dtypes in *exclude*.

**Raises****ValueError**

- If both of *include* and *exclude* are empty

```
>>> df = ks.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df.select_dtypes()
Traceback (most recent call last):
...
ValueError: at least one of include or exclude must be nonempty
```

- If *include* and *exclude* have overlapping elements

```
>>> df = ks.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3})
>>> df.select_dtypes(include='a', exclude='a')
Traceback (most recent call last):
...
ValueError: include and exclude overlap on {'a'}
```

## Notes

- To select datetimes, use `np.datetime64`, `'datetime'` or `'datetime64'`

## Examples

```
>>> df = ks.DataFrame({'a': [1, 2] * 3,
...                    'b': [True, False] * 3,
...                    'c': [1.0, 2.0] * 3,
...                    'd': ['a', 'b'] * 3}, columns=['a', 'b', 'c', 'd'])
>>> df
   a      b      c  d
0  1   True  1.0  a
1  2  False  2.0  b
2  1   True  1.0  a
3  2  False  2.0  b
4  1   True  1.0  a
5  2  False  2.0  b
```

```
>>> df.select_dtypes(include='bool')
   b
0  True
1  False
2  True
3  False
4  True
5  False
```

```
>>> df.select_dtypes(include=['float64'], exclude=['int'])
   c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0
```

```
>>> df.select_dtypes(exclude=['int'])
   b      c  d
0  True  1.0  a
1  False  2.0  b
2  True  1.0  a
3  False  2.0  b
4  True  1.0  a
5  False  2.0  b
```

Spark SQL DDL type strings can be used as well.

```
>>> df.select_dtypes(exclude=['string'])
   a      b      c
0  1    True  1.0
1  2   False  2.0
2  1    True  1.0
3  2   False  2.0
4  1    True  1.0
5  2   False  2.0
```

## databricks.koalas.DataFrame.values

### property DataFrame.values

Return a Numpy representation of the DataFrame or the Series.

**Warning:** We recommend using *DataFrame.to\_numpy()* or *Series.to\_numpy()* instead.

**Note:** This method should only be used if the resulting NumPy ndarray is expected to be small, as all the data is loaded into the driver's memory.

### Returns

**numpy.ndarray**

## Examples

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = ks.DataFrame({'age': [3, 29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0    3     94     31
1   29    170    115
>>> df.dtypes
age      int64
height  int64
weight  int64
dtype: object
>>> df.values
array([[ 3,  94,  31],
       [29, 170, 115]])
```

A DataFrame with mixed type columns (e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = ks.DataFrame([('parrot', 24.0, 'second'),
...                     ('lion', 80.5, 'first'),
...                     ('monkey', np.nan, None)],
```

(continues on next page)

(continued from previous page)

```

...           columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name           object
max_speed      float64
rank           object
dtype: object
>>> df2.values
array([[ 'parrot', 24.0, 'second'],
       [ 'lion', 80.5, 'first'],
       [ 'monkey', nan, None]], dtype=object)

```

For Series,

```

>>> ks.Series([1, 2, 3]).values
array([1, 2, 3])

```

```

>>> ks.Series(list('aabc')).values
array(['a', 'a', 'b', 'c'], dtype=object)

```

### 3.4.3 Conversion

<code>DataFrame.copy([deep])</code>	Make a copy of this object's indices and data.
<code>DataFrame.isna()</code>	Detects missing values for items in the current Dataframe.
<code>DataFrame.astype(dtype)</code>	Cast a Koalas object to a specified dtype dtype.
<code>DataFrame.isnull()</code>	Detects missing values for items in the current Dataframe.
<code>DataFrame.notna()</code>	Detects non-missing values for items in the current Dataframe.
<code>DataFrame.notnull()</code>	Detects non-missing values for items in the current Dataframe.
<code>DataFrame.pad([axis, inplace, limit])</code>	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='ffill'</code> .
<code>DataFrame.bool()</code>	Return the bool of a single element in the current object.

#### **databricks.koalas.DataFrame.copy**

`DataFrame.copy(deep=None) → databricks.koalas.frame.DataFrame`  
 Make a copy of this object's indices and data.

##### **Parameters**

**deep** [None] this parameter is not supported but just dummy parameter to match pandas.

##### **Returns**

**copy** [DataFrame]

## Examples

```
>>> df = ks.DataFrame({'x': [1, 2], 'y': [3, 4], 'z': [5, 6], 'w': [7, 8]},
...                     columns=['x', 'y', 'z', 'w'])
>>> df
   x  y  z  w
0  1  3  5  7
1  2  4  6  8
>>> df_copy = df.copy()
>>> df_copy
   x  y  z  w
0  1  3  5  7
1  2  4  6  8
```

## databricks.koalas.DataFrame.isna

`DataFrame.isna()` → `databricks.koalas.frame.DataFrame`

Detects missing values for items in the current Dataframe.

Return a boolean same-sized Dataframe indicating if the values are NA. NA values, such as None or numpy.NaN, gets mapped to True values. Everything else gets mapped to False values.

**See also:**

[\*DataFrame.notnull\*](#)

## Examples

```
>>> df = ks.DataFrame([(0.2, 0.3), (0.0, None), (0.6, None), (0.2, 0.1)])
>>> df.isnull()
   0      1
0  False False
1  False  True
2  False  True
3  False False
```

```
>>> df = ks.DataFrame([None, 'bee', None], ['dog', None, 'fly'])
>>> df.isnull()
   0      1      2
0  True False  True
1 False  True False
```

## databricks.koalas.DataFrame.astype

`DataFrame.astype(dtype)` → `databricks.koalas.frame.DataFrame`

Cast a Koalas object to a specified dtype dtype.

### Parameters

**dtype** [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire Koalas object to the same type. Alternatively, use {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

**Returns****casted** [same type as caller]**See also:***to\_datetime* Convert argument to datetime.**Examples**

```
>>> df = ks.DataFrame({'a': [1, 2, 3], 'b': [1, 2, 3]}, dtype='int64')
>>> df
   a  b
0  1  1
1  2  2
2  3  3
```

Convert to float type:

```
>>> df.astype('float')
   a  b
0  1.0  1.0
1  2.0  2.0
2  3.0  3.0
```

Convert to int64 type back:

```
>>> df.astype('int64')
   a  b
0  1  1
1  2  2
2  3  3
```

Convert column a to float type:

```
>>> df.astype({'a': float})
   a  b
0  1.0  1
1  2.0  2
2  3.0  3
```

**databricks.koalas.DataFrame.isnull**`DataFrame.isnull()` → `databricks.koalas.frame.DataFrame`

Detects missing values for items in the current Dataframe.

Return a boolean same-sized Dataframe indicating if the values are NA. NA values, such as None or numpy.NaN, gets mapped to True values. Everything else gets mapped to False values.

**See also:***DataFrame.notnull*

## Examples

```
>>> df = ks.DataFrame([(0.2, 0.3), (0.0, None), (0.6, None), (0.2, 0.1)])
>>> df.isnull()
      0      1
0  False  False
1  False   True
2  False   True
3  False  False
```

```
>>> df = ks.DataFrame([None, 'bee', None], ['dog', None, 'fly'])
>>> df.isnull()
      0      1      2
0   True  False   True
1  False   True  False
```

## databricks.koalas.DataFrame.notna

`DataFrame.notna()` → `databricks.koalas.frame.DataFrame`

Detects non-missing values for items in the current Dataframe.

This function takes a dataframe and indicates whether its values are valid (not missing, which is NaN in numeric datatypes, None or NaT in objects and NaT in datetimelike).

**See also:**

*[DataFrame.isnull](#)*

## Examples

```
>>> df = ks.DataFrame([(0.2, 0.3), (0.0, None), (0.6, None), (0.2, 0.1)])
>>> df.notnull()
      0      1
0   True   True
1   True  False
2   True  False
3   True   True
```

```
>>> df = ks.DataFrame(['ant', 'bee', 'cat'], ['dog', None, 'fly'])
>>> df.notnull()
      0      1      2
0   True   True   True
1   True  False   True
```



**databricks.koalas.DataFrame.notnull**

`DataFrame.notnull()` → `databricks.koalas.frame.DataFrame`

Detects non-missing values for items in the current Dataframe.

This function takes a dataframe and indicates whether it's values are valid (not missing, which is NaN in numeric datatypes, None or NaN in objects and NaT in datetimelike).

See also:

[`DataFrame.isnull`](#)

**Examples**

```
>>> df = ks.DataFrame([(0.2, 0.3), (0.0, None), (0.6, None), (0.2, 0.1)])
>>> df.notnull()
   0    1
0  True  True
1  True False
2  True False
3  True  True
```

```
>>> df = ks.DataFrame(['ant', 'bee', 'cat'], ['dog', None, 'fly'])
>>> df.notnull()
   0    1    2
0  True  True  True
1  True False  True
```

**databricks.koalas.DataFrame.pad**

`DataFrame.pad(axis=None, inplace=False, limit=None)` → `Union[DataFrame, Series]`

Synonym for `DataFrame.fillna()` or `Series.fillna()` with `method='ffill'`.

---

**Note:** the current implementation of 'ffill' uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

**Parameters**

**axis** [{0 or index}] 1 and *columns* are not supported.

**inplace** [boolean, default False] Fill in place (do not create a new object)

**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

**Returns**

**DataFrame or Series** DataFrame or Series with NA entries filled.

## Examples

```
>>> kdf = ks.DataFrame({
...     'A': [None, 3, None, None],
...     'B': [2, 4, None, 3],
...     'C': [None, None, None, 1],
...     'D': [0, 1, 5, 4]
...     },
...     columns=['A', 'B', 'C', 'D'])
>>> kdf
```

	A	B	C	D
0	NaN	2.0	NaN	0
1	3.0	4.0	NaN	1
2	NaN	NaN	NaN	5
3	NaN	3.0	1.0	4

Propagate non-null values forward.

```
>>> kdf.ffmpeg()
      A      B      C      D
0  NaN  2.0  NaN  0
1  3.0  4.0  NaN  1
2  3.0  4.0  NaN  5
3  3.0  3.0  1.0  4
```

### For Series

```
>>> kser = ks.Series([2, 4, None, 3])
>>> kser
0      2.0
1      4.0
2      NaN
3      3.0
dtype: float64
```

```
>>> kser.ffmpeg()
0      2.0
1      4.0
2      4.0
3      3.0
dtype: float64
```

## `databricks.koalas.DataFrame.bool`

`DataFrame.bool()` → bool

Return the bool of a single element in the current object.

This must be a boolean scalar value, either True or False. Raise a `ValueError` if the object does not have exactly 1 element, or that element is not boolean

### Returns

bool

## Examples

```
>>> ks.DataFrame({'a': [True]}).bool()
True
```

```
>>> ks.Series([False]).bool()
False
```

If there are non-boolean or multiple values exist, it raises an exception in all cases as below.

```
>>> ks.DataFrame({'a': ['a']}).bool()
Traceback (most recent call last):
...
ValueError: bool cannot act on a non-boolean single element DataFrame
```

```
>>> ks.DataFrame({'a': [True], 'b': [False]}).bool()
Traceback (most recent call last):
...
ValueError: The truth value of a DataFrame is ambiguous. Use a.empty, a.bool(),
a.item(), a.any() or a.all().
```

```
>>> ks.Series([1]).bool()
Traceback (most recent call last):
...
ValueError: bool cannot act on a non-boolean single element DataFrame
```

### 3.4.4 Indexing, iteration

<code>DataFrame.at</code>	Access a single value for a row/column label pair.
<code>DataFrame.iat</code>	Access a single value for a row/column pair by integer position.
<code>DataFrame.head([n])</code>	Return the first $n$ rows.
<code>DataFrame.idxmax([axis])</code>	Return index of first occurrence of maximum over requested axis.
<code>DataFrame.idxmin([axis])</code>	Return index of first occurrence of minimum over requested axis.
<code>DataFrame.loc</code>	Access a group of rows and columns by label(s) or a boolean Series.
<code>DataFrame.iloc</code>	Purely integer-location based indexing for selection by position.
<code>DataFrame.items()</code>	This is an alias of <code>iteritems</code> .
<code>DataFrame.iteritems()</code>	Iterator over (column name, Series) pairs.
<code>DataFrame.iterrows()</code>	Iterate over DataFrame rows as (index, Series) pairs.
<code>DataFrame.itertuples([index, name])</code>	Iterate over DataFrame rows as namedtuples.
<code>DataFrame.keys()</code>	Return alias for columns.
<code>DataFrame.pop(item)</code>	Return item and drop from frame.
<code>DataFrame.tail([n])</code>	Return the last $n$ rows.
<code>DataFrame.xs(key[, axis, level])</code>	Return cross-section from the DataFrame.
<code>DataFrame.get(key[, default])</code>	Get item from object for given key (DataFrame column, Panel slice, etc.).
<code>DataFrame.where(cond[, other])</code>	Replace values where the condition is False.

continues on next page

Table 45 – continued from previous page

<code>DataFrame.mask(cond[, other])</code>	Replace values where the condition is True.
<code>DataFrame.query(expr[, inplace])</code>	Query the columns of a DataFrame with a boolean expression.

**databricks.koalas.DataFrame.at****property** `DataFrame.at`

Access a single value for a row/column label pair. If the index is not unique, all matching pairs are returned as an array. Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get a single value in a DataFrame or Series.

---

**Note:** Unlike pandas, Koalas only allows using `at` to get values but not to set them.

---



---

**Note:** Warning: If `row_index` matches a lot of rows, large amounts of data will be fetched, potentially causing your machine to run out of memory.

---

**Raises**

**KeyError** When label does not exist in DataFrame

**Examples**

```
>>> kdf = ks.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                     index=[4, 5, 5], columns=['A', 'B', 'C'])
>>> kdf
   A  B  C
4  0  2  3
5  0  4  1
5 10 20 30
```

Get value at specified row/column pair

```
>>> kdf.at[4, 'B']
2
```

Get array if an index occurs multiple times

```
>>> kdf.at[5, 'B']
array([ 4, 20])
```

**databricks.koalas.DataFrame.iat****property** `DataFrame.iat`

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

**Raises**

**KeyError** When label does not exist in DataFrame

**Examples**

```
>>> df = ks.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                    columns=['A', 'B', 'C'])
>>> df
   A  B  C
0  0  2  3
1  0  4  1
2 10 20 30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Get value within a series

```
>>> kser = ks.Series([1, 2, 3], index=[10, 20, 30])
>>> kser
10    1
20    2
30    3
dtype: int64
```

```
>>> kser.iat[1]
2
```

**databricks.koalas.DataFrame.head**

`DataFrame.head(n: int = 5) → databricks.koalas.frame.DataFrame`

Return the first  $n$  rows.

This function returns the first  $n$  rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

**Parameters**

**n** [int, default 5] Number of rows to select.

**Returns**

**obj\_head** [same type as caller] The first  $n$  rows of the caller object.

## Examples

```
>>> df = ks.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                               'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
   animal
0  alligator
1      bee
2    falcon
3     lion
4    monkey
5    parrot
6     shark
7     whale
8     zebra
```

Viewing the first 5 lines

```
>>> df.head()
   animal
0  alligator
1      bee
2    falcon
3     lion
4    monkey
```

Viewing the first  $n$  lines (three in this case)

```
>>> df.head(3)
   animal
0  alligator
1      bee
2    falcon
```

## `databricks.koalas.DataFrame.idxmax`

`DataFrame.idxmax` (*axis=0*) → Series

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

---

**Note:** This API collect all rows with maximum value using `to_pandas()` because we suppose the number of rows with max values are usually small in general.

---

### Parameters

**axis** [0 or 'index'] Can only be set to 0 at the moment.

### Returns

Series

See also:

[`Series.idxmax`](#)

## Examples

```
>>> kdf = ks.DataFrame({'a': [1, 2, 3, 2],
...                     'b': [4.0, 2.0, 3.0, 1.0],
...                     'c': [300, 200, 400, 200]})
>>> kdf
   a  b  c
0  1  4.0 300
1  2  2.0 200
2  3  3.0 400
3  2  1.0 200
```

```
>>> kdf.idxmax()
a    2
b    0
c    2
dtype: int64
```

### For Multi-column Index

```
>>> kdf = ks.DataFrame({'a': [1, 2, 3, 2],
...                     'b': [4.0, 2.0, 3.0, 1.0],
...                     'c': [300, 200, 400, 200]})
>>> kdf.columns = pd.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> kdf
   a  b  c
x  y  z
0  1  4.0 300
1  2  2.0 200
2  3  3.0 400
3  2  1.0 200
```

```
>>> kdf.idxmax()
a x    2
b y    0
c z    2
dtype: int64
```

## databricks.koalas.DataFrame.idxmin

`DataFrame.idxmin (axis=0) → Series`

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

**Note:** This API collect all rows with minimum value using `to_pandas()` because we suppose the number of rows with min values are usually small in general.

### Parameters

**axis** [0 or 'index'] Can only be set to 0 at the moment.

### Returns

**Series**

See also:

*Series.idxmin***Examples**

```
>>> kdf = ks.DataFrame({'a': [1, 2, 3, 2],
...                     'b': [4.0, 2.0, 3.0, 1.0],
...                     'c': [300, 200, 400, 200]})
>>> kdf
   a    b    c
0  1  4.0  300
1  2  2.0  200
2  3  3.0  400
3  2  1.0  200
```

```
>>> kdf.idxmin()
a    0
b    3
c    1
dtype: int64
```

**For Multi-column Index**

```
>>> kdf = ks.DataFrame({'a': [1, 2, 3, 2],
...                     'b': [4.0, 2.0, 3.0, 1.0],
...                     'c': [300, 200, 400, 200]})
>>> kdf.columns = pd.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> kdf
   a    b    c
x  y  z
0  1  4.0  300
1  2  2.0  200
2  3  3.0  400
3  2  1.0  200
```

```
>>> kdf.idxmin()
a  x    0
b  y    3
c  z    1
dtype: int64
```

**databricks.koalas.DataFrame.loc****property** `DataFrame.loc`

Access a group of rows and columns by label(s) or a boolean Series.

`.loc[]` is primarily label based, but may also be used with a conditional boolean Series derived from the DataFrame or Series.

Allowed inputs are:

- A single label, e.g. 5 or 'a', (note that 5 is interpreted as a *label* of the index, and **never** as an integer position along the index) for column selection.
- A list or array of labels, e.g. ['a', 'b', 'c'].
- A slice object with labels, e.g. 'a': 'f'.



- A conditional boolean Series derived from the DataFrame or Series
- A boolean array of the same length as the column axis being sliced, e.g. `[True, False, True]`.
- An alignable boolean pandas Series to the column axis being sliced. The index of the key will be aligned before masking.

Not allowed inputs which pandas allows are:

- A boolean array of the same length as the row axis being sliced, e.g. `[True, False, True]`.
- A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

---

**Note:** MultiIndex is not supported yet.

---



---

**Note:** Note that contrary to usual python slices, **both** the start and the stop are included, and the step of the slice is not allowed.

---



---

**Note:** With a list or array of labels for row selection, Koalas behaves as a filter without reordering by the labels.

---

**See also:**

[\*Series.loc\*](#) Access group of values using labels.

## Examples

### Getting values

```
>>> df = ks.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=['cobra', 'viper', 'sidewinder'],
...                    columns=['max_speed', 'shield'])
>>> df
```

	max_speed	shield
cobra	1	2
viper	4	5
sidewinder	7	8

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[ ]]` returns a DataFrame. Also note that Koalas behaves just a filter without reordering by the labels.

```
>>> df.loc[['viper', 'sidewinder']]
```

	max_speed	shield
viper	4	5
sidewinder	7	8

```
>>> df.loc[['sidewinder', 'viper']]
           max_speed  shield
viper              4       5
sidewinder         7       8
```

Single label for column.

```
>>> df.loc['cobra', 'shield']
2
```

List of labels for row.

```
>>> df.loc[['cobra'], 'shield']
cobra      2
Name: shield, dtype: int64
```

List of labels for column.

```
>>> df.loc['cobra', ['shield']]
shield      2
Name: cobra, dtype: int64
```

List of labels for both row and column.

```
>>> df.loc[['cobra'], ['shield']]
           shield
cobra           2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra      1
viper      4
Name: max_speed, dtype: int64
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
           max_speed  shield
sidewinder         7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
           max_speed
sidewinder         7
```

A boolean array of the same length as the column axis being sliced.

```
>>> df.loc[:, [False, True]]
           shield
cobra           2
viper           5
sidewinder       8
```

An alignable boolean Series to the column axis being sliced.

```
>>> df.loc[:, pd.Series([False, True], index=['max_speed', 'shield'])]
      shield
cobra      2
viper      5
sidewinder  8
```

### Setting values

Setting value for all items matching the list of labels.

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
      max_speed  shield
cobra          1      2
viper          4     50
sidewinder     7     50
```

### Setting value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
      max_speed  shield
cobra         10     10
viper          4     50
sidewinder     7     50
```

### Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
      max_speed  shield
cobra         30     10
viper         30     50
sidewinder    30     50
```

### Set value for an entire list of columns

```
>>> df.loc[:, ['max_speed', 'shield']] = 100
>>> df
      max_speed  shield
cobra        100    100
viper        100    100
sidewinder   100    100
```

### Set value with Series

```
>>> df.loc[:, 'shield'] = df['shield'] * 2
>>> df
      max_speed  shield
cobra        100    200
viper        100    200
sidewinder   100    200
```

### Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = ks.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=[7, 8, 9],
...                    columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

## **databricks.koalas.DataFrame.iloc**

### **property** DataFrame.**iloc**

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from 0 to `length-1` of the axis), but may also be used with a conditional boolean Series.

Allowed inputs are:

- An integer for column selection, e.g. 5.
- A list or array of integers for row selection with distinct index values, e.g. [3, 4, 0].
- A list or array of integers for column selection, e.g. [4, 3, 0].
- A boolean array for column selection.
- A slice object with ints for row and column selection, e.g. 1:7.

Not allowed inputs which pandas allows are:

- A list or array of integers for row selection with duplicated indexes, e.g. [4, 4, 0].
- A boolean array for row selection.
- A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

**See also:**

**DataFrame.loc** Purely label-location based indexer for selection by label.

**Series.iloc** Purely integer-location based indexing for selection by position.

## Examples

```
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
>>> df = ks.DataFrame(mydict, columns=['a', 'b', 'c', 'd'])
>>> df
   a     b     c     d
0   1     2     3     4
1 100    200   300   400
2 1000  2000  3000  4000
```

### Indexing just the rows

A scalar integer for row selection.

```
>>> df.iloc[1]
a     100
b     200
c     300
d     400
Name: 1, dtype: int64
```

```
>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
```

With a *slice* object.

```
>>> df.iloc[:3]
   a     b     c     d
0   1     2     3     4
1 100    200   300   400
2 1000  2000  3000  4000
```

### Indexing both axes

You can mix the indexer types for the index and columns. Use `:` to select the entire axis.

With scalar integers.

```
>>> df.iloc[:1, 1]
0     2
Name: b, dtype: int64
```

With lists of integers.

```
>>> df.iloc[:2, [1, 3]]
   b     d
0   2     4
1 200    400
```

With *slice* objects.

```
>>> df.iloc[:2, 0:3]
   a     b     c
0   1     2     3
1 100    200   300
```

With a boolean array whose length matches the columns.

```
>>> df.iloc[:, [True, False, True, False]]
   a      c
0   1      3
1  100    300
2 1000   3000
```

### Setting values

Setting value for all items matching the list of labels.

```
>>> df.iloc[[1, 2], [1]] = 50
>>> df
   a      b      c      d
0   1      2      3      4
1  100   50   300   400
2 1000   50  3000  4000
```

Setting value for an entire row

```
>>> df.iloc[0] = 10
>>> df
   a      b      c      d
0  10   10   10   10
1  100   50   300   400
2 1000   50  3000  4000
```

Set value for an entire column

```
>>> df.iloc[:, 2] = 30
>>> df
   a      b      c      d
0  10   10   30   10
1  100   50   30   400
2 1000   50   30  4000
```

Set value for an entire list of columns

```
>>> df.iloc[:, [2, 3]] = 100
>>> df
   a      b      c      d
0  10   10  100  100
1  100   50  100  100
2 1000   50  100  100
```

Set value with Series

```
>>> df.iloc[:, 3] = df.iloc[:, 3] * 2
>>> df
   a      b      c      d
0  10   10  100  200
1  100   50  100  200
2 1000   50  100  200
```

**databricks.koalas.DataFrame.items**

`DataFrame.items()` → Iterator  
 This is an alias of `iteritems`.

**databricks.koalas.DataFrame.iteritems**

`DataFrame.iteritems()` → Iterator  
 Iterator over (column name, Series) pairs.

Iterates over the DataFrame columns, returning a tuple with the column name and the content as a Series.

**Returns**

**label** [object] The column names for the DataFrame being iterated over.

**content** [Series] The column entries belonging to each label, as a Series.

**Examples**

```
>>> df = ks.DataFrame({'species': ['bear', 'bear', 'marsupial'],
...                    'population': [1864, 22000, 80000]},
...                    index=['panda', 'polar', 'koala'],
...                    columns=['species', 'population'])
>>> df
```

	species	population
panda	bear	1864
polar	bear	22000
koala	marsupial	80000

```
>>> for label, content in df.iteritems():
...     print('label:', label)
...     print('content:', content.to_string())
...
label: species
content: panda          bear
polar          bear
koala  marsupial
label: population
content: panda      1864
polar      22000
koala      80000
```

**databricks.koalas.DataFrame.iterrows**

`DataFrame.iterrows()` → Iterator  
 Iterate over DataFrame rows as (index, Series) pairs.

**Yields**

**index** [label or tuple of label] The index of the row. A tuple for a *MultiIndex*.

**data** [pandas.Series] The data of the row as a Series.

**it** [generator] A generator that iterates over the rows of the frame.

## Notes

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
>>> df = ks.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
>>> print(df['int'].dtype)
int64
```

To preserve dtypes while iterating over the rows, it is better to use `itertuples()` which returns named-tuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

## `databricks.koalas.DataFrame.itertuples`

`DataFrame.itertuples` (*index*: `bool` = `True`, *name*: `Optional[str]` = `'Koalas'`) → Iterator  
Iterate over DataFrame rows as namedtuples.

### Parameters

**index** [`bool`, default `True`] If `True`, return the index as the first element of the tuple.

**name** [`str` or `None`, default `"Koalas"`] The name of the returned namedtuples or `None` to return regular tuples.

### Returns

**iterator** An object to iterate over namedtuples for each row in the DataFrame with the first field possibly being the index and following fields being the column values.

### See also:

`DataFrame.iterrows` Iterate over DataFrame rows as (`index`, `Series`) pairs.

`DataFrame.items` Iterate over (`column name`, `Series`) pairs.

## Notes

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. On python versions < 3.7 regular tuples are returned for DataFrames with a large number of columns (>254).



## Examples

```
>>> df = ks.DataFrame({'num_legs': [4, 2], 'num_wings': [0, 2]},
...                    index=['dog', 'hawk'])
>>> df
```

	num_legs	num_wings
dog	4	0
hawk	2	2

```
>>> for row in df.itertuples():
...     print(row)
...
Koalas(Index='dog', num_legs=4, num_wings=0)
Koalas(Index='hawk', num_legs=2, num_wings=2)
```

By setting the *index* parameter to `False` we can remove the index as the first element of the tuple:

```
>>> for row in df.itertuples(index=False):
...     print(row)
...
Koalas(num_legs=4, num_wings=0)
Koalas(num_legs=2, num_wings=2)
```

With the *name* parameter set we set a custom name for the yielded namedtuples:

```
>>> for row in df.itertuples(name='Animal'):
...     print(row)
...
Animal(Index='dog', num_legs=4, num_wings=0)
Animal(Index='hawk', num_legs=2, num_wings=2)
```

## databricks.koalas.DataFrame.keys

`DataFrame.keys()` → `pandas.core.indexes.base.Index`  
Return alias for columns.

### Returns

**Index** Columns of the DataFrame.

## Examples

```
>>> df = ks.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=['cobra', 'viper', 'sidewinder'],
...                    columns=['max_speed', 'shield'])
>>> df
```

	max_speed	shield
cobra	1	2
viper	4	5
sidewinder	7	8

```
>>> df.keys()
Index(['max_speed', 'shield'], dtype='object')
```

**databricks.koalas.DataFrame.pop**

`DataFrame.pop(item)` → `databricks.koalas.frame.DataFrame`  
 Return item and drop from frame. Raise `KeyError` if not found.

**Parameters**

**item** [str] Label of column to be popped.

**Returns**

**Series**

**Examples**

```
>>> df = ks.DataFrame([('falcon', 'bird', 389.0),
...                     ('parrot', 'bird', 24.0),
...                     ('lion', 'mammal', 80.5),
...                     ('monkey', 'mammal', np.nan)],
...                     columns=('name', 'class', 'max_speed'))
```

```
>>> df
   name  class  max_speed
0  falcon   bird    389.0
1  parrot   bird     24.0
2   lion  mammal     80.5
3  monkey  mammal      NaN
```

```
>>> df.pop('class')
0    bird
1    bird
2  mammal
3  mammal
Name: class, dtype: object
```

```
>>> df
   name  max_speed
0  falcon    389.0
1  parrot     24.0
2   lion     80.5
3  monkey      NaN
```

**Also support for MultiIndex**

```
>>> df = ks.DataFrame([('falcon', 'bird', 389.0),
...                     ('parrot', 'bird', 24.0),
...                     ('lion', 'mammal', 80.5),
...                     ('monkey', 'mammal', np.nan)],
...                     columns=('name', 'class', 'max_speed'))
>>> columns = [('a', 'name'), ('a', 'class'), ('b', 'max_speed')]
>>> df.columns = pd.MultiIndex.from_tuples(columns)
>>> df
      a      b
   name  class max_speed
0  falcon   bird    389.0
1  parrot   bird     24.0
```

(continues on next page)

(continued from previous page)

2	lion	mammal	80.5
3	monkey	mammal	NaN

```
>>> df.pop('a')
      name  class
0  falcon   bird
1  parrot   bird
2    lion  mammal
3  monkey  mammal
```

```
>>> df
      b
max_speed
0    389.0
1     24.0
2     80.5
3      NaN
```

**databricks.koalas.DataFrame.tail**

`DataFrame.tail (n=5) → databricks.koalas.frame.DataFrame`

Return the last  $n$  rows.

This function returns last  $n$  rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

For negative values of  $n$ , this function returns all rows except the first  $n$  rows, equivalent to `df[n:]`.

**Parameters**

**n** [int, default 5] Number of rows to select.

**Returns**

**type of caller** The last  $n$  rows of the caller object.

**See also:**

**`DataFrame.head`** The first  $n$  rows of the caller object.

**Examples**

```
>>> df = ks.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                               'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2    falcon
3      lion
4    monkey
5    parrot
6     shark
7     whale
8     zebra
```

Viewing the last 5 lines

```
>>> df.tail()
      animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last  $n$  lines (three in this case)

```
>>> df.tail(3)
      animal
6   shark
7   whale
8   zebra
```

For negative values of  $n$

```
>>> df.tail(-3)
      animal
3    lion
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

### **databricks.koalas.DataFrame.xs**

`DataFrame.xs(key, axis=0, level=None) → Union[DataFrame, Series]`

Return cross-section from the DataFrame.

This method takes a *key* argument to select data at a particular level of a MultiIndex.

#### **Parameters**

**key** [label or tuple of label] Label contained in the index, or partially in a MultiIndex.

**axis** [0 or 'index', default 0] Axis to retrieve cross-section on. currently only support 0 or 'index'

**level** [object, defaults to first  $n$  levels ( $n=1$  or  $\text{len}(\text{key})$ )] In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

#### **Returns**

**DataFrame or Series** Cross-section from the original DataFrame corresponding to the selected index levels.

**See also:**

**`DataFrame.loc`** Access a group of rows and columns by label(s) or a boolean array.

**`DataFrame.iloc`** Purely integer-location based indexing for selection by position.

## Examples

```
>>> d = {'num_legs': [4, 4, 2, 2],
...      'num_wings': [0, 0, 2, 2],
...      'class': ['mammal', 'mammal', 'mammal', 'bird'],
...      'animal': ['cat', 'dog', 'bat', 'penguin'],
...      'locomotion': ['walks', 'walks', 'flies', 'walks']}
>>> df = ks.DataFrame(data=d)
>>> df = df.set_index(['class', 'animal', 'locomotion'])
>>> df
```

			num_legs	num_wings
class	animal	locomotion		
mammal	cat	walks	4	0
	dog	walks	4	0
	bat	flies	2	2
bird	penguin	walks	2	2

### Get values at specified index

```
>>> df.xs('mammal')
           num_legs  num_wings
animal locomotion
cat     walks      4          0
dog     walks      4          0
bat     flies      2          2
```

### Get values at several indexes

```
>>> df.xs(('mammal', 'dog'))
           num_legs  num_wings
locomotion
walks           4          0
```

```
>>> df.xs(('mammal', 'dog', 'walks'))
num_legs      4
num_wings     0
Name: (mammal, dog, walks), dtype: int64
```

### Get values at specified index and level

```
>>> df.xs('cat', level=1)
           num_legs  num_wings
class locomotion
mammal walks      4          0
```

## databricks.koalas.DataFrame.get

`DataFrame.get(key, default=None) → Any`

Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

### Parameters

**key** [object]

### Returns

**value** [same type as items contained in object]

## Examples

```
>>> df = ks.DataFrame({'x':range(3), 'y':['a','b','b'], 'z':['a','b','b']},
...                     columns=['x', 'y', 'z'], index=[10, 20, 20])
>>> df
   x  y  z
10  0  a  a
20  1  b  b
20  2  b  b
```

```
>>> df.get('x')
10    0
20    1
20    2
Name: x, dtype: int64
```

```
>>> df.get(['x', 'y'])
   x  y
10  0  a
20  1  b
20  2  b
```

```
>>> df.x.get(10)
0
```

```
>>> df.x.get(20)
20    1
20    2
Name: x, dtype: int64
```

```
>>> df.x.get(15, -1)
-1
```

## `databricks.koalas.DataFrame.where`

`DataFrame.where(cond, other=nan) → databricks.koalas.frame.DataFrame`

Replace values where the condition is False.

### Parameters

**cond** [boolean DataFrame] Where cond is True, keep the original value. Where False, replace with corresponding value from other.

**other** [scalar, DataFrame] Entries where cond is False are replaced with corresponding value from other.

### Returns

**DataFrame**

## Examples

```
>>> from databricks.koalas.config import set_option, reset_option
>>> set_option("compute.ops_on_diff_frames", True)
>>> df1 = ks.DataFrame({'A': [0, 1, 2, 3, 4], 'B': [100, 200, 300, 400, 500]})
>>> df2 = ks.DataFrame({'A': [0, -1, -2, -3, -4], 'B': [-100, -200, -300, -400, -
→ 500]})
>>> df1
   A    B
0  0  100
1  1  200
2  2  300
3  3  400
4  4  500
>>> df2
   A    B
0  0 -100
1 -1 -200
2 -2 -300
3 -3 -400
4 -4 -500
```

```
>>> df1.where(df1 > 0).sort_index()
   A    B
0 NaN  100.0
1  1.0  200.0
2  2.0  300.0
3  3.0  400.0
4  4.0  500.0
```

```
>>> df1.where(df1 > 1, 10).sort_index()
   A    B
0  10  100
1  10  200
2   2  300
3   3  400
4   4  500
```

```
>>> df1.where(df1 > 1, df1 + 100).sort_index()
   A    B
0  100  100
1  101  200
2   2  300
3   3  400
4   4  500
```

```
>>> df1.where(df1 > 1, df2).sort_index()
   A    B
0  0  100
1 -1  200
2  2  300
3  3  400
4  4  500
```

When the column name of cond is different from self, it treats all values are False

```
>>> cond = ks.DataFrame({'C': [0, -1, -2, -3, -4], 'D': [4, 3, 2, 1, 0]}) % 3 == 0
>>> cond
      C      D
0  True  False
1 False   True
2 False  False
3  True  False
4 False   True
```

```
>>> df1.where(cond).sort_index()
      A      B
0  NaN  NaN
1  NaN  NaN
2  NaN  NaN
3  NaN  NaN
4  NaN  NaN
```

When the type of `cond` is `Series`, it just check boolean regardless of column name

```
>>> cond = ks.Series([1, 2]) > 1
>>> cond
0    False
1     True
dtype: bool
```

```
>>> df1.where(cond).sort_index()
      A      B
0  NaN   NaN
1  1.0  200.0
2  NaN   NaN
3  NaN   NaN
4  NaN   NaN
```

```
>>> reset_option("compute.ops_on_diff_frames")
```

## **databricks.koalas.DataFrame.mask**

`DataFrame.mask(cond, other=nan) → databricks.koalas.frame.DataFrame`

Replace values where the condition is True.

### **Parameters**

**cond** [boolean DataFrame] Where `cond` is False, keep the original value. Where True, replace with corresponding value from `other`.

**other** [scalar, DataFrame] Entries where `cond` is True are replaced with corresponding value from `other`.

### **Returns**

**DataFrame**



## Examples

```
>>> from databricks.koalas.config import set_option, reset_option
>>> set_option("compute.ops_on_diff_frames", True)
>>> df1 = ks.DataFrame({'A': [0, 1, 2, 3, 4], 'B': [100, 200, 300, 400, 500]})
>>> df2 = ks.DataFrame({'A': [0, -1, -2, -3, -4], 'B': [-100, -200, -300, -400, -
→500]})
>>> df1
   A    B
0  0  100
1  1  200
2  2  300
3  3  400
4  4  500
>>> df2
   A    B
0  0 -100
1 -1 -200
2 -2 -300
3 -3 -400
4 -4 -500
```

```
>>> df1.mask(df1 > 0).sort_index()
   A    B
0  0.0 NaN
1  NaN NaN
2  NaN NaN
3  NaN NaN
4  NaN NaN
```

```
>>> df1.mask(df1 > 1, 10).sort_index()
   A    B
0  0   10
1  1   10
2  10  10
3  10  10
4  10  10
```

```
>>> df1.mask(df1 > 1, df1 + 100).sort_index()
   A    B
0    0  200
1    1  300
2  102  400
3  103  500
4  104  600
```

```
>>> df1.mask(df1 > 1, df2).sort_index()
   A    B
0  0 -100
1  1 -200
2 -2 -300
3 -3 -400
4 -4 -500
```

```
>>> reset_option("compute.ops_on_diff_frames")
```

**databricks.koalas.DataFrame.query**

`DataFrame.query` (*expr*, *inplace=False*) → `Optional[databricks.koalas.frame.DataFrame]`

Query the columns of a DataFrame with a boolean expression.

**Note:** Internal columns that starting with a `'__'` prefix are able to access, however, they are not supposed to be accessed.

**Note:** This API delegates to Spark SQL so the syntax follows Spark SQL. Therefore, the pandas specific syntax such as `@` is not supported. If you want the pandas syntax, you can work around with `DataFrame.koalas.apply_batch()`, but you should be aware that `query_func` will be executed at different nodes in a distributed manner. So, for example, to use `@` syntax, make sure the variable is serialized by, for example, putting it within the closure as below.

```
>>> df = ks.DataFrame({'A': range(2000), 'B': range(2000)})
>>> def query_func(pdf):
...     num = 1995
...     return pdf.query('A > @num')
>>> df.koalas.apply_batch(query_func)
```

	A	B
1996	1996	1996
1997	1997	1997
1998	1998	1998
1999	1999	1999

**Parameters**

**expr** [str] The query string to evaluate.

You can refer to column names that contain spaces by surrounding them in backticks.

For example, if one of your columns is called `a a` and you want to sum it with `b`, your query should be ``a a` + b`.

**inplace** [bool] Whether the query should modify the data in place or return a modified copy.

**Returns**

**DataFrame** DataFrame resulting from the provided query expression.

**Examples**

```
>>> df = ks.DataFrame({'A': range(1, 6),
...                    'B': range(10, 0, -2),
...                    'C C': range(10, 5, -1)})
>>> df
```

	A	B	C C
0	1	10	10
1	2	8	9
2	3	6	8
3	4	4	7
4	5	2	6

```
>>> df.query('A > B')
   A  B  C  C
4  5  2   6
```

The previous expression is equivalent to

```
>>> df[df.A > df.B]
   A  B  C  C
4  5  2   6
```

For columns with spaces in their name, you can use backtick quoting.

```
>>> df.query('B == `C C`')
   A  B  C  C
0  1  10  10
```

The previous expression is equivalent to

```
>>> df[df.B == df['C C']]
   A  B  C  C
0  1  10  10
```

### 3.4.5 Binary operator functions

<code>DataFrame.add(other)</code>	Get Addition of dataframe and other, element-wise (binary operator +).
<code>DataFrame.radd(other)</code>	Get Addition of dataframe and other, element-wise (binary operator +).
<code>DataFrame.div(other)</code>	Get Floating division of dataframe and other, element-wise (binary operator /).
<code>DataFrame.rdiv(other)</code>	Get Floating division of dataframe and other, element-wise (binary operator /).
<code>DataFrame.truediv(other)</code>	Get Floating division of dataframe and other, element-wise (binary operator /).
<code>DataFrame.rtruediv(other)</code>	Get Floating division of dataframe and other, element-wise (binary operator /).
<code>DataFrame.mul(other)</code>	Get Multiplication of dataframe and other, element-wise (binary operator *).
<code>DataFrame.rmul(other)</code>	Get Multiplication of dataframe and other, element-wise (binary operator *).
<code>DataFrame.sub(other)</code>	Get Subtraction of dataframe and other, element-wise (binary operator -).
<code>DataFrame.rsub(other)</code>	Get Subtraction of dataframe and other, element-wise (binary operator -).
<code>DataFrame.pow(other)</code>	Get Exponential power of series of dataframe and other, element-wise (binary operator **).
<code>DataFrame.rpow(other)</code>	Get Exponential power of dataframe and other, element-wise (binary operator **).
<code>DataFrame.mod(other)</code>	Get Modulo of dataframe and other, element-wise (binary operator %).
<code>DataFrame.rmod(other)</code>	Get Modulo of dataframe and other, element-wise (binary operator %).

continues on next page

Table 46 – continued from previous page

<code>DataFrame.floordiv(other)</code>	Get Integer division of dataframe and other, element-wise (binary operator <code>//</code> ).
<code>DataFrame.rfloordiv(other)</code>	Get Integer division of dataframe and other, element-wise (binary operator <code>//</code> ).
<code>DataFrame.lt(other)</code>	Compare if the current value is less than the other.
<code>DataFrame.gt(other)</code>	Compare if the current value is greater than the other.
<code>DataFrame.le(other)</code>	Compare if the current value is less than or equal to the other.
<code>DataFrame.ge(other)</code>	Compare if the current value is greater than or equal to the other.
<code>DataFrame.ne(other)</code>	Compare if the current value is not equal to the other.
<code>DataFrame.eq(other)</code>	Compare if the current value is equal to the other.
<code>DataFrame.dot(other)</code>	Compute the matrix multiplication between the DataFrame and other.

**databricks.koalas.DataFrame.add**

`DataFrame.add(other) → databricks.koalas.frame.DataFrame`

Get Addition of dataframe and other, element-wise (binary operator `+`).

Equivalent to `dataframe + other`. With reverse version, `radd`.

Among flexible wrappers (`add`, `sub`, `mul`, `div`) to arithmetic operators: `+`, `-`, `*`, `/`, `//`.

**Parameters**

**other** [scalar] Any single data

**Returns**

**DataFrame** Result of the arithmetic operation.

**Examples**

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'],
...                    columns=['angles', 'degrees'])
>>> df
```

	angles	degrees
circle	0	360
triangle	3	180
rectangle	4	360

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

(continues on next page)

(continued from previous page)

circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
      angles  degrees
circle      0      720
triangle    6      360
rectangle   8      720
```

```
>>> df + df + df
      angles  degrees
circle      0     1080
triangle     9      540
rectangle   12     1080
```

```
>>> df.radd(1)
      angles  degrees
circle      1      361
triangle     4      181
rectangle     5      361
```

Divide and true divide by constant with reverse version.

```
>>> df / 10
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle    0.4     36.0
```

```
>>> df.div(10)
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle    0.4     36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle    0.4     36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.sub(1)
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.rsub(1)
      angles  degrees
circle         1    -359
triangle        2   -179
rectangle       3   -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle         0     360
triangle        3     180
rectangle        4     360
```

```
>>> df.mul(1)
      angles  degrees
circle         0     360
triangle        3     180
rectangle        4     360
```

```
>>> df.rmul(1)
      angles  degrees
circle         0     360
triangle        3     180
rectangle        4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle      0.0     36.0
triangle    0.0     18.0
rectangle   0.0     36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle      0.0     36.0
triangle    0.0     18.0
rectangle   0.0     36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle      inf      0.0
```

(continues on next page)

(continued from previous page)

triangle	3.0	0.0
rectangle	2.0	0.0

Mod by constant with reverse version.

```
>>> df % 2
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.mod(2)
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.rmod(2)
```

	angles	degrees
circle	NaN	2
triangle	2.0	2
rectangle	2.0	2

Power by constant with reverse version.

```
>>> df ** 2
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.pow(2)
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.rpow(2)
```

	angles	degrees
circle	1.0	2.348543e+108
triangle	8.0	1.532496e+54
rectangle	16.0	2.348543e+108

### databricks.koalas.DataFrame.radd

`DataFrame.radd(other) → databricks.koalas.frame.DataFrame`

Get Addition of dataframe and other, element-wise (binary operator +).

Equivalent to `other + dataframe`. With reverse version, *add*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: +, -, \*, /, //.

#### Parameters

**other** [scalar] Any single data

## Returns

**DataFrame** Result of the arithmetic operation.

## Examples

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                     'degrees': [360, 180, 360]},
...                     index=['circle', 'triangle', 'rectangle'],
...                     columns=['angles', 'degrees'])
>>> df
```

	angles	degrees
circle	0	360
triangle	3	180
rectangle	4	360

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
```

	angles	degrees
circle	0	720
triangle	6	360
rectangle	8	720

```
>>> df + df + df
```

	angles	degrees
circle	0	1080
triangle	9	540
rectangle	12	1080

```
>>> df.radd(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

Divide and true divide by constant with reverse version.

```
>>> df / 10
```

	angles	degrees
circle	0.0	36.0
triangle	0.3	18.0
rectangle	0.4	36.0



```
>>> df.div(10)
      angles  degrees
circle      0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle    3.333333  0.055556
rectangle   2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle      0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle    3.333333  0.055556
rectangle   2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle     -1    359
triangle     2    179
rectangle     3    359
```

```
>>> df.sub(1)
      angles  degrees
circle     -1    359
triangle     2    179
rectangle     3    359
```

```
>>> df.rsub(1)
      angles  degrees
circle      1   -359
triangle    -2   -179
rectangle   -3   -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle      0    360
triangle     3    180
rectangle     4    360
```

```
>>> df.mul(1)
      angles  degrees
circle      0    360
```

(continues on next page)

(continued from previous page)

triangle	3	180
rectangle	4	360

```
>>> df.rmul(1)
      angles  degrees
circle      0     360
triangle    3     180
rectangle   4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle     inf      0.0
triangle   3.0      0.0
rectangle  2.0      0.0
```

Mod by constant with reverse version.

```
>>> df % 2
      angles  degrees
circle      0        0
triangle    1        0
rectangle   0        0
```

```
>>> df.mod(2)
      angles  degrees
circle      0        0
triangle    1        0
rectangle   0        0
```

```
>>> df.rmod(2)
      angles  degrees
circle     NaN        2
triangle   2.0        2
rectangle  2.0        2
```

Power by constant with reverse version.

```
>>> df ** 2
      angles  degrees
circle    0.0 129600.0
```

(continues on next page)

(continued from previous page)

```
triangle      9.0    32400.0
rectangle     16.0   129600.0
```

```
>>> df.pow(2)
           angles  degrees
circle          0.0  129600.0
triangle        9.0   32400.0
rectangle       16.0  129600.0
```

```
>>> df.rpow(2)
           angles  degrees
circle          1.0  2.348543e+108
triangle        8.0   1.532496e+54
rectangle       16.0  2.348543e+108
```

### databricks.koalas.DataFrame.div

`DataFrame.div` (*other*) → `databricks.koalas.frame.DataFrame`

Get Floating division of dataframe and other, element-wise (binary operator /).

Equivalent to `dataframe / other`. With reverse version, *rdiv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: `+`, `-`, `*`, `/`, `//`.

#### Parameters

**other** [scalar] Any single data

#### Returns

**DataFrame** Result of the arithmetic operation.

### Examples

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'],
...                    columns=['angles', 'degrees'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
```

(continues on next page)

(continued from previous page)

circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
      angles  degrees
circle      0      720
triangle    6      360
rectangle   8      720
```

```
>>> df + df + df
      angles  degrees
circle      0     1080
triangle     9      540
rectangle   12     1080
```

```
>>> df.radd(1)
      angles  degrees
circle      1      361
triangle     4      181
rectangle     5      361
```

Divide and true divide by constant with reverse version.

```
>>> df / 10
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle    0.4     36.0
```

```
>>> df.div(10)
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle    0.4     36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle    0.4     36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.sub(1)
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.rsub(1)
      angles  degrees
circle       1    -359
triangle     -2   -179
rectangle    -3   -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

```
>>> df.mul(1)
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

```
>>> df.rmul(1)
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle      0.0    36.0
triangle    0.0    18.0
rectangle   0.0    36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle      0.0    36.0
triangle    0.0    18.0
rectangle   0.0    36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle      inf     0.0
```

(continues on next page)

(continued from previous page)

triangle	3.0	0.0
rectangle	2.0	0.0

Mod by constant with reverse version.

```
>>> df % 2
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.mod(2)
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.rmod(2)
```

	angles	degrees
circle	NaN	2
triangle	2.0	2
rectangle	2.0	2

Power by constant with reverse version.

```
>>> df ** 2
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.pow(2)
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.rpow(2)
```

	angles	degrees
circle	1.0	2.348543e+108
triangle	8.0	1.532496e+54
rectangle	16.0	2.348543e+108

### **databricks.koalas.DataFrame.rdiv**

`DataFrame.rdiv (other) → databricks.koalas.frame.DataFrame`

Get Floating division of dataframe and other, element-wise (binary operator /).

Equivalent to `other / dataframe`. With reverse version, *div*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: `+`, `-`, `*`, `/`, `//`.

#### **Parameters**

**other** [scalar] Any single data

**Returns****DataFrame** Result of the arithmetic operation.**Examples**

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'],
...                    columns=['angles', 'degrees'])
>>> df
```

	angles	degrees
circle	0	360
triangle	3	180
rectangle	4	360

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
```

	angles	degrees
circle	0	720
triangle	6	360
rectangle	8	720

```
>>> df + df + df
```

	angles	degrees
circle	0	1080
triangle	9	540
rectangle	12	1080

```
>>> df.radd(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

Divide and true divide by constant with reverse version.

```
>>> df / 10
```

	angles	degrees
circle	0.0	36.0
triangle	0.3	18.0
rectangle	0.4	36.0

```
>>> df.div(10)
          angles  degrees
circle         0.0    36.0
triangle       0.3    18.0
rectangle      0.4    36.0
```

```
>>> df.rdiv(10)
          angles  degrees
circle         inf  0.027778
triangle    3.333333  0.055556
rectangle    2.500000  0.027778
```

```
>>> df.truediv(10)
          angles  degrees
circle         0.0    36.0
triangle       0.3    18.0
rectangle      0.4    36.0
```

```
>>> df.rtruediv(10)
          angles  degrees
circle         inf  0.027778
triangle    3.333333  0.055556
rectangle    2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
          angles  degrees
circle        -1    359
triangle         2    179
rectangle        3    359
```

```
>>> df.sub(1)
          angles  degrees
circle        -1    359
triangle         2    179
rectangle        3    359
```

```
>>> df.rsub(1)
          angles  degrees
circle         1   -359
triangle        -2   -179
rectangle       -3   -359
```

Multiply by constant with reverse version.

```
>>> df * 1
          angles  degrees
circle         0    360
triangle        3    180
rectangle        4    360
```

```
>>> df.mul(1)
          angles  degrees
circle         0    360
```

(continues on next page)



(continued from previous page)

triangle	3	180
rectangle	4	360

```
>>> df.rmul(1)
      angles  degrees
circle      0     360
triangle    3     180
rectangle   4     360
```

**Floor Divide by constant with reverse version.**

```
>>> df // 10
      angles  degrees
circle    0.0    36.0
triangle  0.0    18.0
rectangle 0.0    36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle    0.0    36.0
triangle  0.0    18.0
rectangle 0.0    36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle    inf     0.0
triangle  3.0     0.0
rectangle 2.0     0.0
```

**Mod by constant with reverse version.**

```
>>> df % 2
      angles  degrees
circle      0         0
triangle    1         0
rectangle   0         0
```

```
>>> df.mod(2)
      angles  degrees
circle      0         0
triangle    1         0
rectangle   0         0
```

```
>>> df.rmod(2)
      angles  degrees
circle    NaN         2
triangle  2.0         2
rectangle 2.0         2
```

**Power by constant with reverse version.**

```
>>> df ** 2
      angles  degrees
circle    0.0 129600.0
```

(continues on next page)

(continued from previous page)

```
triangle      9.0    32400.0
rectangle     16.0   129600.0
```

```
>>> df.pow(2)
           angles  degrees
circle          0.0  129600.0
triangle        9.0   32400.0
rectangle       16.0  129600.0
```

```
>>> df.rpow(2)
           angles  degrees
circle          1.0  2.348543e+108
triangle        8.0   1.532496e+54
rectangle       16.0  2.348543e+108
```

**databricks.koalas.DataFrame.truediv**

`DataFrame.truediv(other)` → `databricks.koalas.frame.DataFrame`

Get Floating division of dataframe and other, element-wise (binary operator /).

Equivalent to `dataframe / other`. With reverse version, `rtruediv`.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: `+`, `-`, `*`, `/`, `//`.

**Parameters**

**other** [scalar] Any single data

**Returns**

**DataFrame** Result of the arithmetic operation.

**Examples**

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                     'degrees': [360, 180, 360]},
...                     index=['circle', 'triangle', 'rectangle'],
...                     columns=['angles', 'degrees'])
>>> df
           angles  degrees
circle          0     360
triangle        3     180
rectangle       4     360
```

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
           angles  degrees
circle          1     361
triangle        4     181
rectangle       5     361
```

```
>>> df.add(1)
           angles  degrees
```

(continues on next page)

(continued from previous page)

circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
      angles  degrees
circle      0      720
triangle    6      360
rectangle   8      720
```

```
>>> df + df + df
      angles  degrees
circle      0     1080
triangle     9      540
rectangle   12     1080
```

```
>>> df.radd(1)
      angles  degrees
circle      1      361
triangle     4      181
rectangle     5      361
```

Divide and true divide by constant with reverse version.

```
>>> df / 10
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle   0.4     36.0
```

```
>>> df.div(10)
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle   0.4     36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle   0.4     36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.sub(1)
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.rsub(1)
      angles  degrees
circle       1    -359
triangle    -2    -179
rectangle   -3    -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

```
>>> df.mul(1)
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

```
>>> df.rmul(1)
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle      0.0    36.0
triangle    0.0    18.0
rectangle   0.0    36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle      0.0    36.0
triangle    0.0    18.0
rectangle   0.0    36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle      inf     0.0
```

(continues on next page)

(continued from previous page)

triangle	3.0	0.0
rectangle	2.0	0.0

Mod by constant with reverse version.

```
>>> df % 2
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.mod(2)
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.rmod(2)
```

	angles	degrees
circle	NaN	2
triangle	2.0	2
rectangle	2.0	2

Power by constant with reverse version.

```
>>> df ** 2
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.pow(2)
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.rpow(2)
```

	angles	degrees
circle	1.0	2.348543e+108
triangle	8.0	1.532496e+54
rectangle	16.0	2.348543e+108

### **databricks.koalas.DataFrame.rtruediv**

`DataFrame.rtruediv(other)` → `databricks.koalas.frame.DataFrame`

Get Floating division of dataframe and other, element-wise (binary operator /).

Equivalent to `other / dataframe`. With reverse version, *truediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: `+`, `-`, `*`, `/`, `//`.

#### **Parameters**

**other** [scalar] Any single data

## Returns

**DataFrame** Result of the arithmetic operation.

## Examples

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'],
...                    columns=['angles', 'degrees'])
>>> df
```

	angles	degrees
circle	0	360
triangle	3	180
rectangle	4	360

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
```

	angles	degrees
circle	0	720
triangle	6	360
rectangle	8	720

```
>>> df + df + df
```

	angles	degrees
circle	0	1080
triangle	9	540
rectangle	12	1080

```
>>> df.radd(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

Divide and true divide by constant with reverse version.

```
>>> df / 10
```

	angles	degrees
circle	0.0	36.0
triangle	0.3	18.0
rectangle	0.4	36.0

```
>>> df.div(10)
      angles  degrees
circle      0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle      0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle     -1    359
triangle    2    179
rectangle   3    359
```

```
>>> df.sub(1)
      angles  degrees
circle     -1    359
triangle    2    179
rectangle   3    359
```

```
>>> df.rsub(1)
      angles  degrees
circle      1   -359
triangle    -2   -179
rectangle   -3   -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle      0    360
triangle    3    180
rectangle   4    360
```

```
>>> df.mul(1)
      angles  degrees
circle      0    360
```

(continues on next page)

(continued from previous page)

triangle	3	180
rectangle	4	360

```
>>> df.rmul(1)
      angles  degrees
circle      0     360
triangle    3     180
rectangle   4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle    inf      0.0
triangle  3.0      0.0
rectangle 2.0      0.0
```

Mod by constant with reverse version.

```
>>> df % 2
      angles  degrees
circle      0         0
triangle    1         0
rectangle   0         0
```

```
>>> df.mod(2)
      angles  degrees
circle      0         0
triangle    1         0
rectangle   0         0
```

```
>>> df.rmod(2)
      angles  degrees
circle    NaN         2
triangle  2.0         2
rectangle 2.0         2
```

Power by constant with reverse version.

```
>>> df ** 2
      angles  degrees
circle    0.0 129600.0
```

(continues on next page)



(continued from previous page)

```
triangle      9.0    32400.0
rectangle     16.0   129600.0
```

```
>>> df.pow(2)
           angles  degrees
circle          0.0  129600.0
triangle        9.0   32400.0
rectangle       16.0  129600.0
```

```
>>> df.rpow(2)
           angles  degrees
circle          1.0  2.348543e+108
triangle        8.0   1.532496e+54
rectangle       16.0  2.348543e+108
```

### databricks.koalas.DataFrame.mul

`DataFrame.mul` (*other*) → `databricks.koalas.frame.DataFrame`

Get Multiplication of dataframe and other, element-wise (binary operator \*).

Equivalent to `dataframe * other`. With reverse version, *rmul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: +, -, \*, /, //.

#### Parameters

**other** [scalar] Any single data

#### Returns

**DataFrame** Result of the arithmetic operation.

### Examples

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                     'degrees': [360, 180, 360]},
...                     index=['circle', 'triangle', 'rectangle'],
...                     columns=['angles', 'degrees'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
```

(continues on next page)

(continued from previous page)

circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
      angles  degrees
circle      0     720
triangle    6     360
rectangle   8     720
```

```
>>> df + df + df
      angles  degrees
circle      0    1080
triangle     9     540
rectangle   12    1080
```

```
>>> df.radd(1)
      angles  degrees
circle      1     361
triangle     4     181
rectangle     5     361
```

Divide and true divide by constant with reverse version.

```
>>> df / 10
      angles  degrees
circle     0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.div(10)
      angles  degrees
circle     0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle     0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.sub(1)
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.rsub(1)
      angles  degrees
circle       1    -359
triangle     -2    -179
rectangle    -3    -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

```
>>> df.mul(1)
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

```
>>> df.rmul(1)
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle      0.0     36.0
triangle    0.0     18.0
rectangle   0.0     36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle      0.0     36.0
triangle    0.0     18.0
rectangle   0.0     36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle      inf     0.0
```

(continues on next page)

(continued from previous page)

triangle	3.0	0.0
rectangle	2.0	0.0

Mod by constant with reverse version.

```
>>> df % 2
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.mod(2)
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.rmod(2)
```

	angles	degrees
circle	NaN	2
triangle	2.0	2
rectangle	2.0	2

Power by constant with reverse version.

```
>>> df ** 2
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.pow(2)
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.rpow(2)
```

	angles	degrees
circle	1.0	2.348543e+108
triangle	8.0	1.532496e+54
rectangle	16.0	2.348543e+108

## databricks.koalas.DataFrame.rmul

`DataFrame.rmul` (*other*) → `databricks.koalas.frame.DataFrame`

Get Multiplication of dataframe and other, element-wise (binary operator \*).

Equivalent to `other * dataframe`. With reverse version, *mul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: +, -, \*, /, //.

### Parameters

**other** [scalar] Any single data

**Returns****DataFrame** Result of the arithmetic operation.**Examples**

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'],
...                    columns=['angles', 'degrees'])
>>> df
```

	angles	degrees
circle	0	360
triangle	3	180
rectangle	4	360

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
```

	angles	degrees
circle	0	720
triangle	6	360
rectangle	8	720

```
>>> df + df + df
```

	angles	degrees
circle	0	1080
triangle	9	540
rectangle	12	1080

```
>>> df.radd(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

Divide and true divide by constant with reverse version.

```
>>> df / 10
```

	angles	degrees
circle	0.0	36.0
triangle	0.3	18.0
rectangle	0.4	36.0

```
>>> df.div(10)
      angles  degrees
circle      0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle      0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle     -1    359
triangle     2    179
rectangle     3    359
```

```
>>> df.sub(1)
      angles  degrees
circle     -1    359
triangle     2    179
rectangle     3    359
```

```
>>> df.rsub(1)
      angles  degrees
circle      1   -359
triangle    -2   -179
rectangle   -3   -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle      0    360
triangle     3    180
rectangle     4    360
```

```
>>> df.mul(1)
      angles  degrees
circle      0    360
```

(continues on next page)

(continued from previous page)

triangle	3	180
rectangle	4	360

```
>>> df.rmul(1)
      angles  degrees
circle      0     360
triangle    3     180
rectangle   4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle    inf     0.0
triangle  3.0     0.0
rectangle 2.0     0.0
```

Mod by constant with reverse version.

```
>>> df % 2
      angles  degrees
circle      0         0
triangle    1         0
rectangle   0         0
```

```
>>> df.mod(2)
      angles  degrees
circle      0         0
triangle    1         0
rectangle   0         0
```

```
>>> df.rmod(2)
      angles  degrees
circle    NaN         2
triangle  2.0         2
rectangle 2.0         2
```

Power by constant with reverse version.

```
>>> df ** 2
      angles  degrees
circle    0.0 129600.0
```

(continues on next page)

(continued from previous page)

triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.pow(2)
      angles  degrees
circle      0.0 129600.0
triangle    9.0  32400.0
rectangle  16.0 129600.0
```

```
>>> df.rpow(2)
      angles  degrees
circle      1.0 2.348543e+108
triangle    8.0  1.532496e+54
rectangle  16.0 2.348543e+108
```

**databricks.koalas.DataFrame.sub**DataFrame.**sub** (*other*) → databricks.koalas.frame.DataFrame

Get Subtraction of dataframe and other, element-wise (binary operator -).

Equivalent to dataframe - other. With reverse version, *rsub*.Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: +, -, \*, /, //.**Parameters****other** [scalar] Any single data**Returns****DataFrame** Result of the arithmetic operation.**Examples**

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'],
...                    columns=['angles', 'degrees'])
>>> df
      angles  degrees
circle      0      360
triangle    3      180
rectangle   4      360
```

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
      angles  degrees
circle      1      361
triangle    4      181
rectangle   5      361
```

```
>>> df.add(1)
      angles  degrees
```

(continues on next page)



(continued from previous page)

circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
      angles  degrees
circle      0      720
triangle    6      360
rectangle   8      720
```

```
>>> df + df + df
      angles  degrees
circle      0     1080
triangle     9      540
rectangle   12     1080
```

```
>>> df.radd(1)
      angles  degrees
circle      1      361
triangle     4      181
rectangle     5      361
```

Divide and true divide by constant with reverse version.

```
>>> df / 10
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle   0.4     36.0
```

```
>>> df.div(10)
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle   0.4     36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle   0.4     36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.sub(1)
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.rsub(1)
      angles  degrees
circle         1    -359
triangle        2   -179
rectangle       3   -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle         0     360
triangle        3     180
rectangle        4     360
```

```
>>> df.mul(1)
      angles  degrees
circle         0     360
triangle        3     180
rectangle        4     360
```

```
>>> df.rmul(1)
      angles  degrees
circle         0     360
triangle        3     180
rectangle        4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle      0.0     36.0
triangle    0.0     18.0
rectangle   0.0     36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle      0.0     36.0
triangle    0.0     18.0
rectangle   0.0     36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle      inf     0.0
```

(continues on next page)

(continued from previous page)

triangle	3.0	0.0
rectangle	2.0	0.0

Mod by constant with reverse version.

```
>>> df % 2
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.mod(2)
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.rmod(2)
```

	angles	degrees
circle	NaN	2
triangle	2.0	2
rectangle	2.0	2

Power by constant with reverse version.

```
>>> df ** 2
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.pow(2)
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.rpow(2)
```

	angles	degrees
circle	1.0	2.348543e+108
triangle	8.0	1.532496e+54
rectangle	16.0	2.348543e+108

### databricks.koalas.DataFrame.rsub

`DataFrame.rsub (other) → databricks.koalas.frame.DataFrame`

Get Subtraction of dataframe and other, element-wise (binary operator -).

Equivalent to `other - dataframe`. With reverse version, *sub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: `+`, `-`, `*`, `/`, `//`.

#### Parameters

**other** [scalar] Any single data

## Returns

**DataFrame** Result of the arithmetic operation.

## Examples

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'],
...                    columns=['angles', 'degrees'])
>>> df
```

	angles	degrees
circle	0	360
triangle	3	180
rectangle	4	360

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
```

	angles	degrees
circle	0	720
triangle	6	360
rectangle	8	720

```
>>> df + df + df
```

	angles	degrees
circle	0	1080
triangle	9	540
rectangle	12	1080

```
>>> df.radd(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

Divide and true divide by constant with reverse version.

```
>>> df / 10
```

	angles	degrees
circle	0.0	36.0
triangle	0.3	18.0
rectangle	0.4	36.0

```
>>> df.div(10)
      angles  degrees
circle      0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle    3.333333  0.055556
rectangle   2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle      0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle    3.333333  0.055556
rectangle   2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle     -1    359
triangle     2    179
rectangle     3    359
```

```
>>> df.sub(1)
      angles  degrees
circle     -1    359
triangle     2    179
rectangle     3    359
```

```
>>> df.rsub(1)
      angles  degrees
circle       1   -359
triangle     -2   -179
rectangle    -3   -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle       0    360
triangle      3    180
rectangle     4    360
```

```
>>> df.mul(1)
      angles  degrees
circle       0    360
```

(continues on next page)

(continued from previous page)

triangle	3	180
rectangle	4	360

```
>>> df.rmul(1)
      angles  degrees
circle      0     360
triangle    3     180
rectangle   4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle    inf      0.0
triangle  3.0      0.0
rectangle 2.0      0.0
```

Mod by constant with reverse version.

```
>>> df % 2
      angles  degrees
circle      0        0
triangle    1        0
rectangle   0        0
```

```
>>> df.mod(2)
      angles  degrees
circle      0        0
triangle    1        0
rectangle   0        0
```

```
>>> df.rmod(2)
      angles  degrees
circle    NaN        2
triangle  2.0        2
rectangle 2.0        2
```

Power by constant with reverse version.

```
>>> df ** 2
      angles  degrees
circle    0.0 129600.0
```

(continues on next page)

(continued from previous page)

```
triangle      9.0    32400.0
rectangle     16.0   129600.0
```

```
>>> df.pow(2)
           angles  degrees
circle          0.0  129600.0
triangle        9.0   32400.0
rectangle       16.0  129600.0
```

```
>>> df.rpow(2)
           angles  degrees
circle          1.0  2.348543e+108
triangle        8.0   1.532496e+54
rectangle       16.0  2.348543e+108
```

### databricks.koalas.DataFrame.pow

`DataFrame.pow(other) → databricks.koalas.frame.DataFrame`

Get Exponential power of series of dataframe and other, element-wise (binary operator `**`).

Equivalent to `dataframe ** other`. With reverse version, `rpow`.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: `+`, `-`, `*`, `/`, `//`.

#### Parameters

**other** [scalar] Any single data

#### Returns

**DataFrame** Result of the arithmetic operation.

### Examples

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'],
...                    columns=['angles', 'degrees'])
>>> df
           angles  degrees
circle          0     360
triangle        3     180
rectangle       4     360
```

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
           angles  degrees
circle          1     361
triangle        4     181
rectangle       5     361
```

```
>>> df.add(1)
           angles  degrees
```

(continues on next page)

(continued from previous page)

circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
      angles  degrees
circle      0      720
triangle    6      360
rectangle   8      720
```

```
>>> df + df + df
      angles  degrees
circle      0     1080
triangle     9      540
rectangle   12     1080
```

```
>>> df.radd(1)
      angles  degrees
circle      1      361
triangle     4      181
rectangle     5      361
```

Divide and true divide by constant with reverse version.

```
>>> df / 10
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle   0.4     36.0
```

```
>>> df.div(10)
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle   0.4     36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle     0.0     36.0
triangle    0.3     18.0
rectangle   0.4     36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract by constant with reverse version.



```
>>> df - 1
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.sub(1)
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.rsub(1)
      angles  degrees
circle       1    -359
triangle     -2    -179
rectangle    -3    -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

```
>>> df.mul(1)
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

```
>>> df.rmul(1)
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle      0.0    36.0
triangle    0.0    18.0
rectangle   0.0    36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle      0.0    36.0
triangle    0.0    18.0
rectangle   0.0    36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle      inf     0.0
```

(continues on next page)

(continued from previous page)

triangle	3.0	0.0
rectangle	2.0	0.0

Mod by constant with reverse version.

```
>>> df % 2
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.mod(2)
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.rmod(2)
```

	angles	degrees
circle	NaN	2
triangle	2.0	2
rectangle	2.0	2

Power by constant with reverse version.

```
>>> df ** 2
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.pow(2)
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.rpow(2)
```

	angles	degrees
circle	1.0	2.348543e+108
triangle	8.0	1.532496e+54
rectangle	16.0	2.348543e+108

## databricks.koalas.DataFrame.rpow

DataFrame.**rpow** (*other*) → databricks.koalas.frame.DataFrame

Get Exponential power of dataframe and other, element-wise (binary operator \*\*).

Equivalent to other \*\* dataframe. With reverse version, *pow*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: +, -, \*, /, //.

### Parameters

**other** [scalar] Any single data

**Returns****DataFrame** Result of the arithmetic operation.**Examples**

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                     'degrees': [360, 180, 360]},
...                     index=['circle', 'triangle', 'rectangle'],
...                     columns=['angles', 'degrees'])
>>> df
```

	angles	degrees
circle	0	360
triangle	3	180
rectangle	4	360

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
```

	angles	degrees
circle	0	720
triangle	6	360
rectangle	8	720

```
>>> df + df + df
```

	angles	degrees
circle	0	1080
triangle	9	540
rectangle	12	1080

```
>>> df.radd(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

Divide and true divide by constant with reverse version.

```
>>> df / 10
```

	angles	degrees
circle	0.0	36.0
triangle	0.3	18.0
rectangle	0.4	36.0

```
>>> df.div(10)
          angles  degrees
circle         0.0    36.0
triangle       0.3    18.0
rectangle      0.4    36.0
```

```
>>> df.rdiv(10)
          angles  degrees
circle         inf  0.027778
triangle    3.333333  0.055556
rectangle    2.500000  0.027778
```

```
>>> df.truediv(10)
          angles  degrees
circle         0.0    36.0
triangle       0.3    18.0
rectangle      0.4    36.0
```

```
>>> df.rtruediv(10)
          angles  degrees
circle         inf  0.027778
triangle    3.333333  0.055556
rectangle    2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
          angles  degrees
circle        -1    359
triangle         2    179
rectangle        3    359
```

```
>>> df.sub(1)
          angles  degrees
circle        -1    359
triangle         2    179
rectangle        3    359
```

```
>>> df.rsub(1)
          angles  degrees
circle         1   -359
triangle        -2   -179
rectangle       -3   -359
```

Multiply by constant with reverse version.

```
>>> df * 1
          angles  degrees
circle         0    360
triangle        3    180
rectangle        4    360
```

```
>>> df.mul(1)
          angles  degrees
circle         0    360
```

(continues on next page)

(continued from previous page)

triangle	3	180
rectangle	4	360

```
>>> df.rmul(1)
      angles  degrees
circle      0      360
triangle    3      180
rectangle   4      360
```

**Floor Divide by constant with reverse version.**

```
>>> df // 10
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle    inf      0.0
triangle  3.0      0.0
rectangle 2.0      0.0
```

**Mod by constant with reverse version.**

```
>>> df % 2
      angles  degrees
circle      0        0
triangle    1        0
rectangle   0        0
```

```
>>> df.mod(2)
      angles  degrees
circle      0        0
triangle    1        0
rectangle   0        0
```

```
>>> df.rmod(2)
      angles  degrees
circle    NaN      2
triangle  2.0      2
rectangle 2.0      2
```

**Power by constant with reverse version.**

```
>>> df ** 2
      angles  degrees
circle    0.0 129600.0
```

(continues on next page)

(continued from previous page)

```
triangle      9.0    32400.0
rectangle     16.0   129600.0
```

```
>>> df.pow(2)
           angles  degrees
circle          0.0  129600.0
triangle        9.0   32400.0
rectangle       16.0  129600.0
```

```
>>> df.rpow(2)
           angles  degrees
circle          1.0  2.348543e+108
triangle        8.0   1.532496e+54
rectangle       16.0  2.348543e+108
```

### **databricks.koalas.DataFrame.mod**

`DataFrame.mod(other) → databricks.koalas.frame.DataFrame`

Get Modulo of dataframe and other, element-wise (binary operator %).

Equivalent to `dataframe % other`. With reverse version, `rmod`.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: `+`, `-`, `*`, `/`, `//`.

#### **Parameters**

**other** [scalar] Any single data

#### **Returns**

**DataFrame** Result of the arithmetic operation.

### **Examples**

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                     'degrees': [360, 180, 360]},
...                     index=['circle', 'triangle', 'rectangle'],
...                     columns=['angles', 'degrees'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
```

(continues on next page)

(continued from previous page)

circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
      angles  degrees
circle      0      720
triangle    6      360
rectangle   8      720
```

```
>>> df + df + df
      angles  degrees
circle      0     1080
triangle     9      540
rectangle   12     1080
```

```
>>> df.radd(1)
      angles  degrees
circle      1      361
triangle     4      181
rectangle     5      361
```

Divide and true divide by constant with reverse version.

```
>>> df / 10
      angles  degrees
circle     0.0     36.0
triangle   0.3     18.0
rectangle  0.4     36.0
```

```
>>> df.div(10)
      angles  degrees
circle     0.0     36.0
triangle   0.3     18.0
rectangle  0.4     36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle     0.0     36.0
triangle   0.3     18.0
rectangle  0.4     36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.sub(1)
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.rsub(1)
      angles  degrees
circle       1    -359
triangle    -2    -179
rectangle   -3    -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

```
>>> df.mul(1)
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

```
>>> df.rmul(1)
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle      0.0    36.0
triangle    0.0    18.0
rectangle   0.0    36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle      0.0    36.0
triangle    0.0    18.0
rectangle   0.0    36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle      inf     0.0
```

(continues on next page)



(continued from previous page)

triangle	3.0	0.0
rectangle	2.0	0.0

Mod by constant with reverse version.

```
>>> df % 2
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.mod(2)
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.rmod(2)
```

	angles	degrees
circle	NaN	2
triangle	2.0	2
rectangle	2.0	2

Power by constant with reverse version.

```
>>> df ** 2
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.pow(2)
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.rpow(2)
```

	angles	degrees
circle	1.0	2.348543e+108
triangle	8.0	1.532496e+54
rectangle	16.0	2.348543e+108

### databricks.koalas.DataFrame.rmod

DataFrame.**rmod**(*other*) → databricks.koalas.frame.DataFrame

Get Modulo of dataframe and other, element-wise (binary operator %).

Equivalent to `other % dataframe`. With reverse version, *mod*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: `+`, `-`, `*`, `/`, `//`.

#### Parameters

**other** [scalar] Any single data

**Returns**

**DataFrame** Result of the arithmetic operation.

**Examples**

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'],
...                    columns=['angles', 'degrees'])
>>> df
```

	angles	degrees
circle	0	360
triangle	3	180
rectangle	4	360

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
```

	angles	degrees
circle	0	720
triangle	6	360
rectangle	8	720

```
>>> df + df + df
```

	angles	degrees
circle	0	1080
triangle	9	540
rectangle	12	1080

```
>>> df.radd(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

Divide and true divide by constant with reverse version.

```
>>> df / 10
```

	angles	degrees
circle	0.0	36.0
triangle	0.3	18.0
rectangle	0.4	36.0

```
>>> df.div(10)
      angles  degrees
circle      0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle    3.333333  0.055556
rectangle   2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle      0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle    3.333333  0.055556
rectangle   2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle     -1    359
triangle     2    179
rectangle     3    359
```

```
>>> df.sub(1)
      angles  degrees
circle     -1    359
triangle     2    179
rectangle     3    359
```

```
>>> df.rsub(1)
      angles  degrees
circle      1   -359
triangle    -2   -179
rectangle   -3   -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle      0    360
triangle     3    180
rectangle     4    360
```

```
>>> df.mul(1)
      angles  degrees
circle      0    360
```

(continues on next page)

(continued from previous page)

triangle	3	180
rectangle	4	360

```
>>> df.rmul(1)
      angles  degrees
circle      0     360
triangle    3     180
rectangle   4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle    inf      0.0
triangle  3.0      0.0
rectangle 2.0      0.0
```

Mod by constant with reverse version.

```
>>> df % 2
      angles  degrees
circle      0        0
triangle    1        0
rectangle   0        0
```

```
>>> df.mod(2)
      angles  degrees
circle      0        0
triangle    1        0
rectangle   0        0
```

```
>>> df.rmod(2)
      angles  degrees
circle    NaN        2
triangle  2.0        2
rectangle 2.0        2
```

Power by constant with reverse version.

```
>>> df ** 2
      angles  degrees
circle    0.0 129600.0
```

(continues on next page)

(continued from previous page)

```
triangle      9.0    32400.0
rectangle     16.0   129600.0
```

```
>>> df.pow(2)
           angles  degrees
circle          0.0   129600.0
triangle        9.0    32400.0
rectangle       16.0   129600.0
```

```
>>> df.rpow(2)
           angles  degrees
circle          1.0  2.348543e+108
triangle        8.0  1.532496e+54
rectangle       16.0  2.348543e+108
```

### **databricks.koalas.DataFrame.floordiv**

`DataFrame.floordiv(other) → databricks.koalas.frame.DataFrame`

Get Integer division of dataframe and other, element-wise (binary operator `//`).

Equivalent to `dataframe // other`. With reverse version, `rfloordiv`.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: `+`, `-`, `*`, `/`, `//`.

#### **Parameters**

**other** [scalar] Any single data

#### **Returns**

**DataFrame** Result of the arithmetic operation.

### **Examples**

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                     'degrees': [360, 180, 360]},
...                     index=['circle', 'triangle', 'rectangle'],
...                     columns=['angles', 'degrees'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
```

(continues on next page)

(continued from previous page)

circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
      angles  degrees
circle      0      720
triangle    6      360
rectangle   8      720
```

```
>>> df + df + df
      angles  degrees
circle      0     1080
triangle     9      540
rectangle   12     1080
```

```
>>> df.radd(1)
      angles  degrees
circle      1      361
triangle     4      181
rectangle     5      361
```

Divide and true divide by constant with reverse version.

```
>>> df / 10
      angles  degrees
circle     0.0     36.0
triangle   0.3     18.0
rectangle  0.4     36.0
```

```
>>> df.div(10)
      angles  degrees
circle     0.0     36.0
triangle   0.3     18.0
rectangle  0.4     36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle     0.0     36.0
triangle   0.3     18.0
rectangle  0.4     36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle  3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.sub(1)
      angles  degrees
circle      -1     359
triangle     2     179
rectangle    3     359
```

```
>>> df.rsub(1)
      angles  degrees
circle       1    -359
triangle     -2    -179
rectangle    -3    -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

```
>>> df.mul(1)
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

```
>>> df.rmul(1)
      angles  degrees
circle       0     360
triangle     3     180
rectangle    4     360
```

Floor Divide by constant with reverse version.

```
>>> df // 10
      angles  degrees
circle      0.0     36.0
triangle    0.0     18.0
rectangle   0.0     36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle      0.0     36.0
triangle    0.0     18.0
rectangle   0.0     36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle      inf     0.0
```

(continues on next page)

(continued from previous page)

triangle	3.0	0.0
rectangle	2.0	0.0

Mod by constant with reverse version.

```
>>> df % 2
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.mod(2)
```

	angles	degrees
circle	0	0
triangle	1	0
rectangle	0	0

```
>>> df.rmod(2)
```

	angles	degrees
circle	NaN	2
triangle	2.0	2
rectangle	2.0	2

Power by constant with reverse version.

```
>>> df ** 2
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.pow(2)
```

	angles	degrees
circle	0.0	129600.0
triangle	9.0	32400.0
rectangle	16.0	129600.0

```
>>> df.rpow(2)
```

	angles	degrees
circle	1.0	2.348543e+108
triangle	8.0	1.532496e+54
rectangle	16.0	2.348543e+108

## databricks.koalas.DataFrame.rfloordiv

`DataFrame.rfloordiv(other) → databricks.koalas.frame.DataFrame`

Get Integer division of dataframe and other, element-wise (binary operator `//`).

Equivalent to `other // dataframe`. With reverse version, *floordiv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*) to arithmetic operators: `+`, `-`, `*`, `/`, `//`.

### Parameters

**other** [scalar] Any single data



**Returns****DataFrame** Result of the arithmetic operation.**Examples**

```
>>> df = ks.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                    index=['circle', 'triangle', 'rectangle'],
...                    columns=['angles', 'degrees'])
>>> df
```

	angles	degrees
circle	0	360
triangle	3	180
rectangle	4	360

Add a scalar with operator version which return the same results. Also reverse version.

```
>>> df + 1
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

```
>>> df.add(df)
```

	angles	degrees
circle	0	720
triangle	6	360
rectangle	8	720

```
>>> df + df + df
```

	angles	degrees
circle	0	1080
triangle	9	540
rectangle	12	1080

```
>>> df.radd(1)
```

	angles	degrees
circle	1	361
triangle	4	181
rectangle	5	361

Divide and true divide by constant with reverse version.

```
>>> df / 10
```

	angles	degrees
circle	0.0	36.0
triangle	0.3	18.0
rectangle	0.4	36.0

```
>>> df.div(10)
      angles  degrees
circle      0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rdiv(10)
      angles  degrees
circle      inf  0.027778
triangle    3.333333  0.055556
rectangle   2.500000  0.027778
```

```
>>> df.truediv(10)
      angles  degrees
circle      0.0    36.0
triangle    0.3    18.0
rectangle   0.4    36.0
```

```
>>> df.rtruediv(10)
      angles  degrees
circle      inf  0.027778
triangle    3.333333  0.055556
rectangle   2.500000  0.027778
```

Subtract by constant with reverse version.

```
>>> df - 1
      angles  degrees
circle     -1    359
triangle     2    179
rectangle     3    359
```

```
>>> df.sub(1)
      angles  degrees
circle     -1    359
triangle     2    179
rectangle     3    359
```

```
>>> df.rsub(1)
      angles  degrees
circle      1   -359
triangle    -2   -179
rectangle   -3   -359
```

Multiply by constant with reverse version.

```
>>> df * 1
      angles  degrees
circle      0    360
triangle     3    180
rectangle     4    360
```

```
>>> df.mul(1)
      angles  degrees
circle      0    360
```

(continues on next page)

(continued from previous page)

triangle	3	180
rectangle	4	360

```
>>> df.rmul(1)
      angles  degrees
circle      0     360
triangle    3     180
rectangle   4     360
```

**Floor Divide by constant with reverse version.**

```
>>> df // 10
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.floordiv(10)
      angles  degrees
circle    0.0     36.0
triangle  0.0     18.0
rectangle 0.0     36.0
```

```
>>> df.rfloordiv(10)
      angles  degrees
circle    inf      0.0
triangle  3.0      0.0
rectangle 2.0      0.0
```

**Mod by constant with reverse version.**

```
>>> df % 2
      angles  degrees
circle      0        0
triangle    1        0
rectangle   0        0
```

```
>>> df.mod(2)
      angles  degrees
circle      0        0
triangle    1        0
rectangle   0        0
```

```
>>> df.rmod(2)
      angles  degrees
circle    NaN      2
triangle  2.0      2
rectangle 2.0      2
```

**Power by constant with reverse version.**

```
>>> df ** 2
      angles  degrees
circle    0.0 129600.0
```

(continues on next page)

(continued from previous page)

```
triangle      9.0    32400.0
rectangle     16.0   129600.0
```

```
>>> df.pow(2)
           angles  degrees
circle         0.0   129600.0
triangle        9.0    32400.0
rectangle      16.0   129600.0
```

```
>>> df.rpow(2)
           angles  degrees
circle         1.0  2.348543e+108
triangle        8.0  1.532496e+54
rectangle      16.0  2.348543e+108
```

### databricks.koalas.DataFrame.lt

`DataFrame.lt(other)` → `databricks.koalas.frame.DataFrame`

Compare if the current value is less than the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                     'b': [1, np.nan, 1, np.nan]},
...                     index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.lt(1)
           a      b
a  False  False
b  False  False
c  False  False
d  False  False
```

### databricks.koalas.DataFrame.gt

`DataFrame.gt(other)` → `databricks.koalas.frame.DataFrame`

Compare if the current value is greater than the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                     'b': [1, np.nan, 1, np.nan]},
...                     index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.gt(2)
           a      b
a  False  False
b  False  False
c   True  False
d   True  False
```

**databricks.koalas.DataFrame.le**

`DataFrame.le(other) → databricks.koalas.frame.DataFrame`

Compare if the current value is less than or equal to the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                    'b': [1, np.nan, 1, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.le(2)
      a      b
a  True  True
b  True False
c False  True
d False False
```

**databricks.koalas.DataFrame.ge**

`DataFrame.ge(other) → databricks.koalas.frame.DataFrame`

Compare if the current value is greater than or equal to the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                    'b': [1, np.nan, 1, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.ge(1)
      a      b
a  True  True
b  True False
c  True  True
d  True False
```

**databricks.koalas.DataFrame.ne**

`DataFrame.ne(other) → databricks.koalas.frame.DataFrame`

Compare if the current value is not equal to the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                    'b': [1, np.nan, 1, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.ne(1)
      a      b
a False False
b  True  True
c  True False
d  True  True
```

**databricks.koalas.DataFrame.eq**

`DataFrame.eq(other) → databricks.koalas.frame.DataFrame`

Compare if the current value is equal to the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                    'b': [1, np.nan, 1, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.eq(1)
      a      b
a  True  True
b False False
c False  True
d False False
```

**databricks.koalas.DataFrame.dot**

`DataFrame.dot(other: Series) → Series`

Compute the matrix multiplication between the DataFrame and other.

This method computes the matrix product between the DataFrame and the values of an other Series

It can also be called using `self @ other` in Python `>= 3.5`.

---

**Note:** This method is based on an expensive operation due to the nature of big data. Internally it needs to generate each row for each value, and then group twice - it is a huge operation. To prevent misuse, this method has the 'compute.max\_rows' default limit of input length, and raises a `ValueError`.

```
>>> from databricks.koalas.config import option_context
>>> with option_context(
...     'compute.max_rows', 1000, "compute.ops_on_diff_frames", True
... ):
...     kdf = ks.DataFrame({'a': range(1001)})
...     kser = ks.Series([2], index=['a'])
...     kdf.dot(kser)
Traceback (most recent call last):
...
ValueError: Current DataFrame has more then the given limit 1000 rows.
Please set 'compute.max_rows' by using 'databricks.koalas.config.set_option'
to retrieve to retrieve more than 1000 rows. Note that, before changing the
'compute.max_rows', this operation is considerably expensive.
```

---

**Parameters**

**other** [Series] The other object to compute the matrix product with.

**Returns**

**Series** Return the matrix product between self and other as a Series.

**See also:**

[`Series.dot`](#) Similar method for Series.

## Notes

The dimensions of DataFrame and other must be compatible in order to compute the matrix multiplication. In addition, the column names of DataFrame and the index of other must contain the same values, as they will be aligned prior to the multiplication.

The dot method for Series computes the inner product, instead of the matrix product here.

## Examples

```
>>> from databricks.koalas.config import set_option, reset_option
>>> set_option("compute.ops_on_diff_frames", True)
>>> kdf = ks.DataFrame([[0, 1, -2, -1], [1, 1, 1, 1]])
>>> kser = ks.Series([1, 1, 2, 1])
>>> kdf.dot(kser)
0    -4
1     5
dtype: int64
```

Note how shuffling of the objects does not change the result.

```
>>> kser2 = kser.reindex([1, 0, 2, 3])
>>> kdf.dot(kser2)
0    -4
1     5
dtype: int64
>>> kdf @ kser2
0    -4
1     5
dtype: int64
>>> reset_option("compute.ops_on_diff_frames")
```

## 3.4.6 Function application, GroupBy & Window

<code>DataFrame.apply(func[, axis, args])</code>	Apply a function along an axis of the DataFrame.
<code>DataFrame.applymap(func)</code>	Apply a function to a Dataframe elementwise.
<code>DataFrame.pipe(func, *args, **kwargs)</code>	Apply func(self, *args, **kwargs).
<code>DataFrame.agg(func)</code>	Aggregate using one or more operations over the specified axis.
<code>DataFrame.aggregate(func)</code>	Aggregate using one or more operations over the specified axis.
<code>DataFrame.groupby(by[, axis, as_index, dropna])</code>	Group DataFrame or Series using a Series of columns.
<code>DataFrame.rolling(window[, min_periods])</code>	Provide rolling transformations.
<code>DataFrame.expanding([min_periods])</code>	Provide expanding transformations.
<code>DataFrame.transform(func[, axis])</code>	Call func on self producing a Series with transformed values and that has the same length as its input.
<code>DataFrame.map_in_pandas(func)</code>	Apply a function that takes pandas DataFrame and outputs pandas DataFrame.

**databricks.koalas.DataFrame.apply**

`DataFrame.apply(func, axis=0, args=(), **kwargs) → Union[Series, DataFrame, Index]`

Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`).

See also [Transform and apply a function](#).

**Note:** when `axis` is 0 or 'index', the `func` is unable to access to the whole input series. Koalas internally splits the input series into multiple batches and calls `func` with each batch multiple times. Therefore, operations such as global aggregations are impossible. See the example below.

```
>>> # This case does not return the length of whole series but of the batch_
↳internally
... # used.
... def length(s) -> int:
...     return len(s)
...
>>> df = ks.DataFrame({'A': range(1000)})
>>> df.apply(length, axis=0)
0      83
1      83
2      83
...
10     83
11     83
dtype: int32
```

**Note:** this API executes the function once to infer the type which is potentially expensive, for instance, when the dataset is created after aggregations or sorting.

To avoid this, specify the return type as *Series* or scalar value in `func`, for instance, as below:

```
>>> def square(s) -> ks.Series[np.int32]:
...     return s ** 2
```

Koalas uses return type hint and does not try to infer the type.

In case when `axis` is 1, it requires to specify *DataFrame* or scalar value with type hints as below:

```
>>> def plus_one(x) -> ks.DataFrame[float, float]:
...     return x + 1
```

If the return type is specified as *DataFrame*, the output column names become `c0`, `c1`, `c2` ... `cn`. These names are positionally mapped to the returned DataFrame in `func`.

To specify the column names, you can assign them in a pandas friendly style as below:

```
>>> def plus_one(x) -> ks.DataFrame["a": float, "b": float]:
...     return x + 1
```

```
>>> pdf = pd.DataFrame({'a': [1, 2, 3], 'b': [3, 4, 5]})
>>> def plus_one(x) -> ks.DataFrame[zip(pdf.dtypes, pdf.columns)]:
...     return x + 1
```



However, this way switches the index type to default index type in the output because the type hint cannot express the index type at this moment. Use `reset_index()` to keep index as a workaround.

When the given function has the return type annotated, the original index of the DataFrame will be lost and then a default index will be attached to the result. Please be careful about configuring the default index. See also [Default Index Type](#).

### Parameters

**func** [function] Function to apply to each column or row.

**axis** [{0 or 'index', 1 or 'columns'}, default 0] Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

**args** [tuple] Positional arguments to pass to *func* in addition to the array/series.

**\*\*kwargs** Additional keyword arguments to pass as keywords arguments to *func*.

### Returns

**Series or DataFrame** Result of applying *func* along the given axis of the DataFrame.

See also:

[`DataFrame.applymap`](#) For elementwise operations.

[`DataFrame.aggregate`](#) Only perform aggregating type operations.

[`DataFrame.transform`](#) Only perform transforming type operations.

[`Series.apply`](#) The equivalent function for Series.

### Examples

```
>>> df = ks.DataFrame([[4, 9]] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> def sqrt(x) -> ks.Series[float]:
...     return np.sqrt(x)
...
>>> df.apply(sqrt, axis=0)
   A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

You can omit the type hint and let Koalas infer its type.

```
>>> df.apply(np.sqrt, axis=0)
      A      B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

When *axis* is 1 or 'columns', it applies the function for each row.

```
>>> def summation(x) -> np.int64:
...     return np.sum(x)
...
>>> df.apply(summation, axis=1)
0    13
1    13
2    13
dtype: int64
```

Likewise, you can omit the type hint and let Koalas infer its type.

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

```
>>> df.apply(max, axis=1)
0     9
1     9
2     9
dtype: int64
```

Returning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

In order to specify the types when *axis* is '1', it should use `DataFrame[...]` annotation. In this case, the column names are automatically generated.

```
>>> def identify(x) -> ks.DataFrame['A': np.int64, 'B': np.int64]:
...     return x
...
>>> df.apply(identify, axis=1)
      A  B
0  4  9
1  4  9
2  4  9
```

You can also specify extra arguments.

```
>>> def plus_two(a, b, c) -> ks.DataFrame[np.int64, np.int64]:
...     return a + b + c
...
>>> df.apply(plus_two, axis=1, args=(1,), c=3)
```

(continues on next page)

(continued from previous page)

	c0	c1
0	8	13
1	8	13
2	8	13

**databricks.koalas.DataFrame.applymap**

`DataFrame.applymap(func) → databricks.koalas.frame.DataFrame`

Apply a function to a Dataframe elementwise.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

**Note:** this API executes the function once to infer the type which is potentially expensive, for instance, when the dataset is created after aggregations or sorting.

To avoid this, specify return type in `func`, for instance, as below:

```
>>> def square(x) -> np.int32:
...     return x ** 2
```

Koalas uses return type hint and does not try to infer the type.

**Parameters**

**func** [callable] Python function, returns a single value from a single value.

**Returns**

**DataFrame** Transformed DataFrame.

**Examples**

```
>>> df = ks.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
   0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> def str_len(x) -> int:
...     return len(str(x))
>>> df.applymap(str_len)
   0  1
0  3  4
1  5  5
```

```
>>> def power(x) -> float:
...     return x ** 2
>>> df.applymap(power)
   0      1
0  1.000000  4.494400
1 11.262736 20.857489
```

You can omit the type hint and let Koalas infer its type.

```
>>> df.applymap(lambda x: x ** 2)
      0      1
0  1.000000  4.494400
1 11.262736 20.857489
```

### `databricks.koalas.DataFrame.pipe`

`DataFrame.pipe(func, *args, **kwargs) → Any`  
 Apply `func(self, *args, **kwargs)`.

#### Parameters

**func** [function] function to apply to the DataFrame. `args`, and `kwargs` are passed into `func`. Alternatively a (callable, data\_keyword) tuple where `data_keyword` is a string indicating the keyword of callable that expects the DataFrames.

**args** [iterable, optional] positional arguments passed into `func`.

**kwargs** [mapping, optional] a dictionary of keyword arguments passed into `func`.

#### Returns

**object** [the return type of `func`.]

### Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. For example, given

```
>>> df = ks.DataFrame({'category': ['A', 'A', 'B'],
...                   'col1': [1, 2, 3],
...                   'col2': [4, 5, 6]},
...                   columns=['category', 'col1', 'col2'])
>>> def keep_category_a(df):
...     return df[df['category'] == 'A']
>>> def add_one(df, column):
...     return df.assign(col3=df[column] + 1)
>>> def multiply(df, column1, column2):
...     return df.assign(col4=df[column1] * df[column2])
```

instead of writing

```
>>> multiply(add_one(keep_category_a(df), column="col1"), column1="col2", column2=
↪ "col3")
  category  col1  col2  col3  col4
0        A     1     4     2     8
1        A     2     5     3    15
```

You can write

```
>>> (df.pipe(keep_category_a)
...   .pipe(add_one, column="col1")
...   .pipe(multiply, column1="col2", column2="col3")
... )
  category  col1  col2  col3  col4
0        A     1     4     2     8
1        A     2     5     3    15
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `df`:

```
>>> def multiply_2(column1, df, column2):
...     return df.assign(col4=df[column1] * df[column2])
```

Then you can write

```
>>> (df.pipe(keep_category_a)
...   .pipe(add_one, column="col1")
...   .pipe((multiply_2, 'df'), column1="col2", column2="col3")
... )
  category  col1  col2  col3  col4
0         A     1     4     2     8
1         A     2     5     3    15
```

You can use lambda as well

```
>>> ks.Series([1, 2, 3]).pipe(lambda x: (x + 1).rename("value"))
0     2
1     3
2     4
Name: value, dtype: int64
```

## **databricks.koalas.DataFrame.agg**

`DataFrame.agg(func: Union[List[str], Dict[Any, List[str]]]) → Union[Series, DataFrame, Index]`

Aggregate using one or more operations over the specified axis.

### **Parameters**

**func** [dict or a list] a dict mapping from column name (string) to aggregate functions (list of strings). If a list is given, the aggregation is performed against all columns.

### **Returns**

**DataFrame**

See also:

**DataFrame.apply** Invoke function on DataFrame.

**DataFrame.transform** Only perform transforming type operations.

**DataFrame.groupby** Perform operations over groups.

**Series.aggregate** The equivalent function for Series.

### **Notes**

`agg` is an alias for `aggregate`. Use the alias.

## Examples

```
>>> df = ks.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

```
>>> df
   A    B    C
0  1.0  2.0  3.0
1  4.0  5.0  6.0
2  7.0  8.0  9.0
3  NaN  NaN  NaN
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])[['A', 'B', 'C']].sort_index()
      A    B    C
min  1.0  2.0  3.0
sum 12.0 15.0 18.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})[['A', 'B']].sort_index()
      A    B
max  NaN  8.0
min   1.0  2.0
sum 12.0  NaN
```

For multi-index columns:

```
>>> df.columns = pd.MultiIndex.from_tuples([("X", "A"), ("X", "B"), ("Y", "C")])
>>> df.agg(['sum', 'min'])[(("X", "A"), ("X", "B"), ("Y", "C"))].sort_index()
      X      Y
      A    B    C
min  1.0  2.0  3.0
sum 12.0 15.0 18.0
```

```
>>> aggregated = df.agg({'X', "A") : ['sum', 'min'], ("X", "B") : ['min', 'max']})
↪
>>> aggregated[(("X", "A"), ("X", "B"))].sort_index()
      X
      A    B
max  NaN  8.0
min   1.0  2.0
sum 12.0  NaN
```

**databricks.koalas.DataFrame.aggregate**

`DataFrame.aggregate` (*func*: Union[List[str], Dict[Any, List[str]]]) → Union[Series, DataFrame, Index]  
 Aggregate using one or more operations over the specified axis.

**Parameters**

**func** [dict or a list] a dict mapping from column name (string) to aggregate functions (list of strings). If a list is given, the aggregation is performed against all columns.

**Returns**

**DataFrame**

See also:

**DataFrame.apply** Invoke function on DataFrame.

**DataFrame.transform** Only perform transforming type operations.

**DataFrame.groupby** Perform operations over groups.

**Series.aggregate** The equivalent function for Series.

**Notes**

*agg* is an alias for *aggregate*. Use the alias.

**Examples**

```
>>> df = ks.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

```
>>> df
   A    B    C
0  1.0  2.0  3.0
1  4.0  5.0  6.0
2  7.0  8.0  9.0
3  NaN  NaN  NaN
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])[['A', 'B', 'C']].sort_index()
      A    B    C
min  1.0  2.0  3.0
sum 12.0 15.0 18.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})[['A', 'B']].sort_index()
      A    B
max  NaN  8.0
min  1.0  2.0
sum 12.0  NaN
```

For multi-index columns:

```
>>> df.columns = pd.MultiIndex.from_tuples([("X", "A"), ("X", "B"), ("Y", "C")])
>>> df.agg(['sum', 'min'])[[("X", "A"), ("X", "B"), ("Y", "C")]].sort_index()
```

	X		Y
	A	B	C
min	1.0	2.0	3.0
sum	12.0	15.0	18.0

```
>>> aggregated = df.agg({"X", "A") : ['sum', 'min'], ("X", "B") : ['min', 'max']}
↳)
>>> aggregated[[("X", "A"), ("X", "B")]].sort_index()
```

	X	
	A	B
max	NaN	8.0
min	1.0	2.0
sum	12.0	NaN

## databricks.koalas.DataFrame.groupby

`DataFrame.groupby` (*by*, *axis=0*, *as\_index: bool = True*, *dropna: bool = True*) →  
 Union[`DataFrameGroupBy`, `SeriesGroupBy`]  
 Group `DataFrame` or `Series` using a `Series` of columns.

A `groupby` operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

### Parameters

**by** [Series, label, or list of labels] Used to determine the groups for the `groupby`. If `Series` is passed, the `Series` or dict `VALUES` will be used to determine the groups. A label or list of labels may be passed to group by the columns in `self`.

**axis** [int, default 0 or 'index'] Can only be set to 0 at the moment.

**as\_index** [bool, default True] For aggregated output, return object with group labels as the index. Only relevant for `DataFrame` input. `as_index=False` is effectively “SQL-style” grouped output.

**dropna** [bool, default True] If True, and if group keys contain NA values, NA values together with row/column will be dropped. If False, NA values will also be treated as the key in groups.

### Returns

**`DataFrameGroupBy` or `SeriesGroupBy`** Depends on the calling object and returns `groupby` object that contains information about the groups.

See also:

`koalas.groupby.GroupBy`



## Examples

```
>>> df = ks.DataFrame({'Animal': ['Falcon', 'Falcon',
...                               'Parrot', 'Parrot'],
...                    'Max Speed': [380., 370., 24., 26.]},
...                    columns=['Animal', 'Max Speed'])
>>> df
   Animal  Max Speed
0  Falcon    380.0
1  Falcon    370.0
2  Parrot     24.0
3  Parrot     26.0
```

```
>>> df.groupby(['Animal']).mean().sort_index()
   Max Speed
Animal
Falcon    375.0
Parrot     25.0
```

```
>>> df.groupby(['Animal'], as_index=False).mean().sort_values('Animal')
...
   Animal  Max Speed
...Falcon    375.0
...Parrot     25.0
```

We can also choose to include NA in group keys or not by setting dropna parameter, the default setting is True:

```
>>> l = [[1, 2, 3], [1, None, 4], [2, 1, 3], [1, 2, 2]]
>>> df = ks.DataFrame(l, columns=["a", "b", "c"])
>>> df.groupby(by=["b"]).sum().sort_index()
   a  c
b
1.0  2  3
2.0  2  5
```

```
>>> df.groupby(by=["b"], dropna=False).sum().sort_index()
   a  c
b
1.0  2  3
2.0  2  5
NaN  1  4
```

## databricks.koalas.DataFrame.rolling

`DataFrame.rolling(window, min_periods=None)` → `databricks.koalas.window.Rolling`  
Provide rolling transformations.

---

**Note:** 'min\_periods' in Koalas works as a fixed window size unlike pandas. Unlike pandas, NA is also counted as the period. This might be changed in the near future.

---

### Parameters

**window** [int, or offset] Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

**min\_periods** [int, default None] Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, min\_periods will default to 1. Otherwise, min\_periods will default to the size of the window.

#### Returns

a Window sub-classed for the particular operation

### **databricks.koalas.DataFrame.expanding**

DataFrame.**expanding** (*min\_periods=1*) → databricks.koalas.window.Expanding  
Provide expanding transformations.

---

**Note:** 'min\_periods' in Koalas works as a fixed window size unlike pandas. Unlike pandas, NA is also counted as the period. This might be changed in the near future.

---

#### Parameters

**min\_periods** [int, default 1] Minimum number of observations in window required to have a value (otherwise result is NA).

#### Returns

a Window sub-classed for the particular operation

### **databricks.koalas.DataFrame.transform**

DataFrame.**transform** (*func, axis=0, \*args, \*\*kwargs*) → databricks.koalas.frame.DataFrame  
Call *func* on self producing a Series with transformed values and that has the same length as its input.  
See also [Transform and apply a function](#).

---

**Note:** this API executes the function once to infer the type which is potentially expensive, for instance, when the dataset is created after aggregations or sorting.

To avoid this, specify return type in *func*, for instance, as below:

```
>>> def square(x) -> ks.Series[np.int32]:  
...     return x ** 2
```

Koalas uses return type hint and does not try to infer the type.

---

---

**Note:** the series within *func* is actually multiple pandas series as the segments of the whole Koalas series; therefore, the length of each series is not guaranteed. As an example, an aggregation against each series does work as a global aggregation but an aggregation of each segment. See below:

```
>>> def func(x) -> ks.Series[np.int32]:  
...     return x + sum(x)
```

#### Parameters

**func** [function] Function to use for transforming the data. It must work when pandas Series is passed.

**axis** [int, default 0 or 'index'] Can only be set to 0 at the moment.

**\*args** Positional arguments to pass to func.

**\*\*kwargs** Keyword arguments to pass to func.

### Returns

**DataFrame** A DataFrame that must have the same length as self.

### Raises

**Exception** [If the returned DataFrame has a different length than self.]

### See also:

**DataFrame.aggregate** Only perform aggregating type operations.

**DataFrame.apply** Invoke function on DataFrame.

**Series.transform** The equivalent function for Series.

### Examples

```
>>> df = ks.DataFrame({'A': range(3), 'B': range(1, 4)}, columns=['A', 'B'])
>>> df
   A  B
0  0  1
1  1  2
2  2  3
```

```
>>> def square(x) -> ks.Series[np.int32]:
...     return x ** 2
>>> df.transform(square)
   A  B
0  0  1
1  1  4
2  4  9
```

You can omit the type hint and let Koalas infer its type.

```
>>> df.transform(lambda x: x ** 2)
   A  B
0  0  1
1  1  4
2  4  9
```

For multi-index columns:

```
>>> df.columns = [('X', 'A'), ('X', 'B')]
>>> df.transform(square)
   X
   A  B
0  0  1
1  1  4
2  4  9
```

```
>>> (df * -1).transform(abs)
      X
      A  B
0    0  1
1    1  2
2    2  3
```

You can also specify extra arguments.

```
>>> def calculation(x, y, z) -> ks.Series[int]:
...     return x ** y + z
>>> df.transform(calculation, y=10, z=20)
      X
      A  B
0    20  21
1    21 1044
2 1044 59069
```

### `databricks.koalas.DataFrame.map_in_pandas`

`DataFrame.map_in_pandas(func) → databricks.koalas.frame.DataFrame`

Apply a function that takes pandas DataFrame and outputs pandas DataFrame. The pandas DataFrame given to the function is of a batch used internally.

See also [Transform and apply a function](#).

**Note:** the *func* is unable to access to the whole input frame. Koalas internally splits the input series into multiple batches and calls *func* with each batch multiple times. Therefore, operations such as global aggregations are impossible. See the example below.

```
>>> # This case does not return the length of whole frame but of the batch_
↳internally
... # used.
... def length(pdf) -> ks.DataFrame[int]:
...     return pd.DataFrame([len(pdf)])
...
>>> df = ks.DataFrame({'A': range(1000)})
>>> df.koalas.apply_batch(length)
      c0
0     83
1     83
2     83
...
10    83
11    83
```

**Note:** this API executes the function once to infer the type which is potentially expensive, for instance, when the dataset is created after aggregations or sorting.

To avoid this, specify return type in *func*, for instance, as below:

```
>>> def plus_one(x) -> ks.DataFrame[float, float]:
...     return x + 1
```

If the return type is specified, the output column names become *c0*, *c1*, *c2* ... *cn*. These names are positionally mapped to the returned DataFrame in *func*.

To specify the column names, you can assign them in a pandas friendly style as below:

```
>>> def plus_one(x) -> ks.DataFrame["a": float, "b": float]:
...     return x + 1
```

```
>>> pdf = pd.DataFrame({'a': [1, 2, 3], 'b': [3, 4, 5]})
>>> def plus_one(x) -> ks.DataFrame[zip(pdf.dtypes, pdf.columns)]:
...     return x + 1
```

When the given function has the return type annotated, the original index of the DataFrame will be lost and a default index will be attached to the result DataFrame. Please be careful about configuring the default index. See also [Default Index Type](#).

### Parameters

**func** [function] Function to apply to each pandas frame.

**args** [tuple] Positional arguments to pass to *func* in addition to the array/series.

**\*\*kwargs** Additional keyword arguments to pass as keywords arguments to *func*.

### Returns

**DataFrame**

See also:

[\*DataFrame.apply\*](#) For row/columnwise operations.

[\*DataFrame.applymap\*](#) For elementwise operations.

[\*DataFrame.aggregate\*](#) Only perform aggregating type operations.

[\*DataFrame.transform\*](#) Only perform transforming type operations.

[\*Series.koalas.transform\\_batch\*](#) transform the search as each pandas chunks.

### Examples

```
>>> df = ks.DataFrame([(1, 2), (3, 4), (5, 6)], columns=['A', 'B'])
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> def query_func(pdf) -> ks.DataFrame[int, int]:
...     return pdf.query('A == 1')
>>> df.koalas.apply_batch(query_func)
   c0  c1
0   1   2
```

```
>>> def query_func(pdf) -> ks.DataFrame["A": int, "B": int]:
...     return pdf.query('A == 1')
>>> df.koalas.apply_batch(query_func)
   A  B
0  1  2
```

You can also omit the type hints so Koalas infers the return schema as below:

```
>>> df.koalas.apply_batch(lambda pdf: pdf.query('A == 1'))
   A  B
0  1  2
```

You can also specify extra arguments.

```
>>> def calculation(pdf, y, z) -> ks.DataFrame[int, int]:
...     return pdf ** y + z
>>> df.koalas.apply_batch(calculation, args=(10,), z=20)
   c0      c1
0    21   1044
1  59069 1048596
2 9765645 60466196
```

You can also use `np.ufunc` and built-in functions as input.

```
>>> df.koalas.apply_batch(np.add, args=(10,))
   A  B
0  11 12
1  13 14
2  15 16
```

```
>>> (df * -1).koalas.apply_batch(abs)
   A  B
0  1  2
1  3  4
2  5  6
```

### 3.4.7 Computations / Descriptive Stats

<code>DataFrame.abs()</code>	Return a Series/DataFrame with absolute numeric value of each element.
<code>DataFrame.all([axis])</code>	Return whether all elements are True.
<code>DataFrame.any([axis])</code>	Return whether any element is True.
<code>DataFrame.clip([lower, upper])</code>	Trim values at input threshold(s).
<code>DataFrame.corr([method])</code>	Compute pairwise correlation of columns, excluding NA/null values.
<code>DataFrame.count([axis, numeric_only])</code>	Count non-NA cells for each column.
<code>DataFrame.describe([percentiles])</code>	Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
<code>DataFrame.kurt([axis, numeric_only])</code>	Return unbiased kurtosis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>DataFrame.kurtosis([axis, numeric_only])</code>	Return unbiased kurtosis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).

continues on next page

Table 48 – continued from previous page

<code>DataFrame.mad([axis])</code>	Return the mean absolute deviation of values.
<code>DataFrame.max([axis, numeric_only])</code>	Return the maximum of the values.
<code>DataFrame.mean([axis, numeric_only])</code>	Return the mean of the values.
<code>DataFrame.min([axis, numeric_only])</code>	Return the minimum of the values.
<code>DataFrame.median([axis, numeric_only, accuracy])</code>	Return the median of the values for the requested axis.
<code>DataFrame.pct_change([periods])</code>	Percentage change between the current and a prior element.
<code>DataFrame.prod([axis, numeric_only, min_count])</code>	Return the product of the values.
<code>DataFrame.product([axis, numeric_only, ...])</code>	Return the product of the values.
<code>DataFrame.quantile([q, axis, numeric_only, ...])</code>	Return value at the given quantile.
<code>DataFrame.nunique([axis, dropna, approx, rsd])</code>	Return number of unique elements in the object.
<code>DataFrame.sem([axis, ddof, numeric_only])</code>	Return unbiased standard error of the mean over requested axis.
<code>DataFrame.skew([axis, numeric_only])</code>	Return unbiased skew normalized by N-1.
<code>DataFrame.sum([axis, numeric_only, min_count])</code>	Return the sum of the values.
<code>DataFrame.std([axis, ddof, numeric_only])</code>	Return sample standard deviation.
<code>DataFrame.var([axis, ddof, numeric_only])</code>	Return unbiased variance.
<code>DataFrame.cummin([skipna])</code>	Return cumulative minimum over a DataFrame or Series axis.
<code>DataFrame.cummax([skipna])</code>	Return cumulative maximum over a DataFrame or Series axis.
<code>DataFrame.cumsum([skipna])</code>	Return cumulative sum over a DataFrame or Series axis.
<code>DataFrame.cumprod([skipna])</code>	Return cumulative product over a DataFrame or Series axis.
<code>DataFrame.round([decimals])</code>	Round a DataFrame to a variable number of decimal places.
<code>DataFrame.diff([periods, axis])</code>	First discrete difference of element.
<code>DataFrame.eval(expr[, inplace])</code>	Evaluate a string describing operations on DataFrame columns.

**databricks.koalas.DataFrame.abs**

`DataFrame.abs()` → Union[DataFrame, Series]

Return a Series/DataFrame with absolute numeric value of each element.

**Returns**

**abs** [Series/DataFrame containing the absolute value of each element.]

**Examples**

Absolute numeric values in a Series.

```
>>> s = ks.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a DataFrame.

```
>>> df = ks.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... },
...     columns=['a', 'b', 'c'])
>>> df.abs()
   a  b  c
0  4 10 100
1  5 20  50
2  6 30  30
3  7 40  50
```

## `databricks.koalas.DataFrame.all`

`DataFrame.all (axis: Union[int, str] = 0) → Series`

Return whether all elements are True.

Returns True unless there is at least one element within a series that is False or equivalent (e.g. zero or empty)

### Parameters

**axis** [{0 or 'index'}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.

### Returns

Series

## Examples

Create a dataframe from a dictionary.

```
>>> df = ks.DataFrame({
...     'col1': [True, True, True],
...     'col2': [True, False, False],
...     'col3': [0, 0, 0],
...     'col4': [1, 2, 3],
...     'col5': [True, True, None],
...     'col6': [True, False, None]},
...     columns=['col1', 'col2', 'col3', 'col4', 'col5', 'col6'])
```

Default behaviour checks if column-wise values all return a boolean.

```
>>> df.all()
col1      True
col2     False
col3     False
col4      True
col5      True
col6     False
dtype: bool
```



**databricks.koalas.DataFrame.any**

`DataFrame.any` (*axis*: *Union[int, str] = 0*) → *Series*

Return whether any element is True.

Returns False unless there is at least one element within a series that is True or equivalent (e.g. non-zero or non-empty).

**Parameters**

**axis** `[[0 or 'index'], default 0]` Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.

**Returns**

**Series**

**Examples**

Create a dataframe from a dictionary.

```
>>> df = ks.DataFrame({
...     'col1': [False, False, False],
...     'col2': [True, False, False],
...     'col3': [0, 0, 1],
...     'col4': [0, 1, 2],
...     'col5': [False, False, None],
...     'col6': [True, False, None]},
...     columns=['col1', 'col2', 'col3', 'col4', 'col5', 'col6'])
```

Default behaviour checks if column-wise values all return a boolean.

```
>>> df.any()
col1    False
col2     True
col3     True
col4     True
col5    False
col6     True
dtype: bool
```

**databricks.koalas.DataFrame.clip**

`DataFrame.clip` (*lower*: *Union[float, int] = None*, *upper*: *Union[float, int] = None*) → *databricks.koalas.frame.DataFrame*

Trim values at input threshold(s).

Assigns values outside boundary to boundary values.

**Parameters**

**lower** `[float or int, default None]` Minimum threshold value. All values below this threshold will be set to it.

**upper** `[float or int, default None]` Maximum threshold value. All values above this threshold will be set to it.

**Returns**

**DataFrame** DataFrame with the values outside the clip boundaries replaced.

### Notes

One difference between this implementation and pandas is that running `pd.DataFrame({'A': ['a', 'b']}).clip(0, 1)` will crash with “TypeError: ‘<=’ not supported between instances of ‘str’ and ‘int’” while `ks.DataFrame({'A': ['a', 'b']}).clip(0, 1)` will output the original DataFrame, simply ignoring the incompatible types.

### Examples

```
>>> ks.DataFrame({'A': [0, 2, 4]}).clip(1, 3)
   A
0  1
1  2
2  3
```

### `databricks.koalas.DataFrame.corr`

`DataFrame.corr(method='pearson') → Union[Series, DataFrame, Index]`  
Compute pairwise correlation of columns, excluding NA/null values.

#### Parameters

**method** [{‘pearson’, ‘spearman’}]

- `pearson` : standard correlation coefficient
- `spearman` : Spearman rank correlation

#### Returns

`y` [DataFrame]

See also:

[\*Series.corr\*](#)

### Notes

There are behavior differences between Koalas and pandas.

- the *method* argument only accepts ‘pearson’, ‘spearman’
- the data should not contain NaNs. Koalas will return an error.
- Koalas doesn’t support the following argument(s).
  - *min\_periods* argument is not supported

## Examples

```
>>> df = ks.DataFrame([(0.2, 0.3), (0.0, 0.6), (0.6, 0.0), (0.2, 0.1)],
...                    columns=['dogs', 'cats'])
>>> df.corr('pearson')
           dogs      cats
dogs  1.000000 -0.851064
cats -0.851064  1.000000
```

```
>>> df.corr('spearman')
           dogs      cats
dogs  1.000000 -0.948683
cats -0.948683  1.000000
```

## databricks.koalas.DataFrame.count

`DataFrame.count` (*axis*: `Union[int, str] = None`, *numeric\_only*: `bool = False`) → `Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]`

Count non-NA cells for each column.

The values *None*, *NaN* are considered NA.

### Parameters

**axis** [{0 or 'index', 1 or 'columns'}], default 0] If 0 or 'index' counts are generated for each column. If 1 or 'columns' counts are generated for each row.

**numeric\_only** [bool, default False] If True, include only float, int, boolean columns. This parameter is mainly for pandas compatibility.

### Returns

**max** [scalar for a Series, and a Series for a DataFrame.]

See also:

**`DataFrame.shape`** Number of DataFrame rows and columns (including NA elements).

**`DataFrame.isna`** Boolean same-sized DataFrame showing places of NA elements.

## Examples

Constructing DataFrame from a dictionary:

```
>>> df = ks.DataFrame({"Person":
...                    ["John", "Myla", "Lewis", "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]},
...                    columns=["Person", "Age", "Single"])
>>> df
  Person  Age  Single
0   John  24.0   False
1   Myla   NaN    True
2  Lewis  21.0    True
3   John  33.0    True
4   Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person      5
Age         4
Single      5
dtype: int64
```

```
>>> df.count(axis=1)
0      3
1      2
2      3
3      3
4      3
dtype: int64
```

On a Series:

```
>>> df['Person'].count()
5
```

```
>>> df['Age'].count()
4
```

## **databricks.koalas.DataFrame.describe**

`DataFrame.describe` (*percentiles: Optional[List[float]] = None*) → `databricks.koalas.frame.DataFrame`

Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

### **Parameters**

**percentiles** [list of `float` in range [0.0, 1.0], default [0.25, 0.5, 0.75]] A list of percentiles to be computed.

### **Returns**

**DataFrame** Summary statistics of the Dataframe provided.

See also:

**`DataFrame.count`** Count number of non-NA/null observations.

**`DataFrame.max`** Maximum of the values in the object.

**`DataFrame.min`** Minimum of the values in the object.

**`DataFrame.mean`** Mean of the values.

**`DataFrame.std`** Standard deviation of the observations.

## Notes

For numeric data, the result's index will include count, mean, std, min, 25%, 50%, 75%, max.

Currently only numeric data is supported.

## Examples

Describing a numeric Series.

```
>>> s = ks.Series([1, 2, 3])
>>> s.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.0
50%        2.0
75%        3.0
max        3.0
dtype: float64
```

Describing a DataFrame. Only numeric fields are returned.

```
>>> df = ks.DataFrame({'numeric1': [1, 2, 3],
...                    'numeric2': [4.0, 5.0, 6.0],
...                    'object': ['a', 'b', 'c']
...                    },
...                    columns=['numeric1', 'numeric2', 'object'])
>>> df.describe()
      numeric1  numeric2
count         3.0      3.0
mean          2.0      5.0
std           1.0      1.0
min           1.0      4.0
25%           1.0      4.0
50%           2.0      5.0
75%           3.0      6.0
max           3.0      6.0
```

For multi-index columns:

```
>>> df.columns = [('num', 'a'), ('num', 'b'), ('obj', 'c')]
>>> df.describe()
      num
      a    b
count  3.0  3.0
mean   2.0  5.0
std    1.0  1.0
min    1.0  4.0
25%    1.0  4.0
50%    2.0  5.0
75%    3.0  6.0
max    3.0  6.0
```

```
>>> df[['num', 'b']].describe()
count      3.0
```

(continues on next page)

(continued from previous page)

```

mean      5.0
std       1.0
min       4.0
25%      4.0
50%      5.0
75%      6.0
max       6.0
Name: (num, b), dtype: float64

```

Describing a DataFrame and selecting custom percentiles.

```

>>> df = ks.DataFrame({'numeric1': [1, 2, 3],
...                     'numeric2': [4.0, 5.0, 6.0]
...                     },
...                     columns=['numeric1', 'numeric2'])
>>> df.describe(percentiles = [0.85, 0.15])

```

	numeric1	numeric2
count	3.0	3.0
mean	2.0	5.0
std	1.0	1.0
min	1.0	4.0
15%	1.0	4.0
50%	2.0	5.0
85%	3.0	6.0
max	3.0	6.0

Describing a column from a DataFrame by accessing it as an attribute.

```

>>> df.numeric1.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.0
50%        2.0
75%        3.0
max        3.0
Name: numeric1, dtype: float64

```

Describing a column from a DataFrame by accessing it as an attribute and selecting custom percentiles.

```

>>> df.numeric1.describe(percentiles = [0.85, 0.15])
count      3.0
mean       2.0
std        1.0
min        1.0
15%        1.0
50%        2.0
85%        3.0
max        3.0
Name: numeric1, dtype: float64

```

**databricks.koalas.DataFrame.kurt**

`DataFrame.kurt` (*axis*: `Union[int, str] = None`, *numeric\_only*: `bool = None`) → `Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]`

Return unbiased kurtosis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

**Parameters**

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**Returns**

**kurt** [scalar for a Series, and a Series for a DataFrame.]

**Examples**

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                    columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.kurtosis()
a    -1.5
b    -1.5
dtype: float64
```

On a Series:

```
>>> df['a'].kurtosis()
-1.5
```

**databricks.koalas.DataFrame.kurtosis**

`DataFrame.kurtosis` (*axis*: `Union[int, str] = None`, *numeric\_only*: `bool = None`) → `Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]`

Return unbiased kurtosis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1.

**Parameters**

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**Returns**

**kurt** [scalar for a Series, and a Series for a DataFrame.]

## Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                     columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.kurtosis()
a    -1.5
b    -1.5
dtype: float64
```

On a Series:

```
>>> df['a'].kurtosis()
-1.5
```

## databricks.koalas.DataFrame.mad

DataFrame.**mad** (*axis=0*) → Series

Return the mean absolute deviation of values.

### Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

## Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                     columns=['a', 'b'])
```

```
>>> df.mad()
a    0.666667
b    0.066667
dtype: float64
```

```
>>> df.mad(axis=1)
0    0.45
1    0.90
2    1.35
3     NaN
dtype: float64
```

## databricks.koalas.DataFrame.max

DataFrame.**max** (*axis: Union[int, str] = None, numeric\_only: bool = None*) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]

Return the maximum of the values.

### Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.



**numeric\_only** [bool, default None] If True, include only float, int, boolean columns. This parameter is mainly for pandas compatibility. False is supported; however, the columns should be all numeric or all non-numeric.

### Returns

**max** [scalar for a Series, and a Series for a DataFrame.]

### Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                    columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.max()
a    3.0
b    0.3
dtype: float64
```

```
>>> df.max(axis=1)
0    1.0
1    2.0
2    3.0
3    NaN
dtype: float64
```

On a Series:

```
>>> df['a'].max()
3.0
```

## databricks.koalas.DataFrame.mean

DataFrame.**mean** (axis: Union[int, str] = None, numeric\_only: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]

Return the mean of the values.

### Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

### Returns

**mean** [scalar for a Series, and a Series for a DataFrame.]

## Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                     columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.mean()
a      2.0
b      0.2
dtype: float64
```

```
>>> df.mean(axis=1)
0      0.55
1      1.10
2      1.65
3       NaN
dtype: float64
```

On a Series:

```
>>> df['a'].mean()
2.0
```

## databricks.koalas.DataFrame.min

`DataFrame.min(axis: Union[int, str] = None, numeric_only: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]`

Return the minimum of the values.

### Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] If True, include only float, int, boolean columns. This parameter is mainly for pandas compatibility. False is supported; however, the columns should be all numeric or all non-numeric.

### Returns

**min** [scalar for a Series, and a Series for a DataFrame.]

## Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                     columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.min()
a      1.0
b      0.1
dtype: float64
```

```
>>> df.min(axis=1)
0    0.1
1    0.2
2    0.3
3    NaN
dtype: float64
```

On a Series:

```
>>> df['a'].min()
1.0
```

### databricks.koalas.DataFrame.median

`DataFrame.median` (*axis: Union[int, str] = None, numeric\_only: bool = None, accuracy: int = 10000*) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]

Return the median of the values for the requested axis.

**Note:** Unlike pandas', the median in Koalas is an approximated median based upon approximate percentile computation because computing median across a large dataset is extremely expensive.

#### Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**accuracy** [int, optional] Default accuracy of approximation. Larger value means better accuracy. The relative error can be deduced by  $1.0 / \text{accuracy}$ .

#### Returns

**median** [scalar or Series]

### Examples

```
>>> df = ks.DataFrame({
...     'a': [24., 21., 25., 33., 26.], 'b': [1, 2, 3, 4, 5]}, columns=['a', 'b'])
>>> df
   a  b
0 24.0 1
1 21.0 2
2 25.0 3
3 33.0 4
4 26.0 5
```

On a DataFrame:

```
>>> df.median()
a    25.0
b     3.0
dtype: float64
```

On a Series:

```
>>> df['a'].median()
25.0
>>> (df['b'] + 100).median()
103.0
```

For multi-index columns,

```
>>> df.columns = pd.MultiIndex.from_tuples([('x', 'a'), ('y', 'b')])
>>> df
      x  y
    a  b
0  24.0  1
1  21.0  2
2  25.0  3
3  33.0  4
4  26.0  5
```

On a DataFrame:

```
>>> df.median()
x  a    25.0
y  b     3.0
dtype: float64
```

```
>>> df.median(axis=1)
0    12.5
1    11.5
2    14.0
3    18.5
4    15.5
dtype: float64
```

On a Series:

```
>>> df[('x', 'a')].median()
25.0
>>> (df[('y', 'b')] + 100).median()
103.0
```

## **databricks.koalas.DataFrame.pct\_change**

`DataFrame.pct_change` (*periods=1*) → `databricks.koalas.frame.DataFrame`

Percentage change between the current and a prior element.

---

**Note:** the current implementation of this API uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

### **Parameters**

**periods** [int, default 1] Periods to shift for forming percent change.

### **Returns**

**DataFrame**

## Examples

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = ks.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
```

	FR	GR	IT
1980-01-01	4.0405	1.7246	804.74
1980-02-01	4.0963	1.7482	810.01
1980-03-01	4.3149	1.8519	860.13

```
>>> df.pct_change()
```

	FR	GR	IT
1980-01-01	NaN	NaN	NaN
1980-02-01	0.013810	0.013684	0.006549
1980-03-01	0.053365	0.059318	0.061876

You can set periods to shift for forming percent change

```
>>> df.pct_change(2)
```

	FR	GR	IT
1980-01-01	NaN	NaN	NaN
1980-02-01	NaN	NaN	NaN
1980-03-01	0.067912	0.073814	0.06883

## databricks.koalas.DataFrame.prod

`DataFrame.prod(axis: Union[int, str] = None, numeric_only: bool = None, min_count: int = 0) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]`  
Return the product of the values.

**Note:** unlike pandas', Koalas' emulates product by `exp(sum(log(...)))` trick. Therefore, it only works for positive numbers.

### Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**min\_count** [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

## Examples

On a DataFrame:

Non-numeric type column is not included to the result.

```
>>> kdf = ks.DataFrame({'A': [1, 2, 3, 4, 5],
...                     'B': [10, 20, 30, 40, 50],
...                     'C': ['a', 'b', 'c', 'd', 'e']})
>>> kdf
   A  B  C
0  1 10  a
1  2 20  b
2  3 30  c
3  4 40  d
4  5 50  e
```

```
>>> kdf.prod()
A      120
B  12000000
dtype: int64
```

If there is no numeric type columns, returns empty Series.

```
>>> ks.DataFrame({"key": ['a', 'b', 'c'], "val": ['x', 'y', 'z']}).prod()
Series([], dtype: float64)
```

On a Series:

```
>>> ks.Series([1, 2, 3, 4, 5]).prod()
120
```

By default, the product of an empty or all-NA Series is 1

```
>>> ks.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> ks.Series([]).prod(min_count=1)
nan
```

## `databricks.koalas.DataFrame.product`

`DataFrame.product` (*axis*: `Union[int, str] = None`, *numeric\_only*: `bool = None`, *min\_count*: `int = 0`) → `Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]`

Return the product of the values.

---

**Note:** unlike pandas', Koalas' emulates product by `exp(sum(log(...)))` trick. Therefore, it only works for positive numbers.

---

### Parameters

**axis** `[[index (0), columns (1)]]` Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**min\_count** [int, default 0] The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

## Examples

On a DataFrame:

Non-numeric type column is not included to the result.

```
>>> kdf = ks.DataFrame({'A': [1, 2, 3, 4, 5],
...                     'B': [10, 20, 30, 40, 50],
...                     'C': ['a', 'b', 'c', 'd', 'e']})
>>> kdf
   A  B  C
0  1 10  a
1  2 20  b
2  3 30  c
3  4 40  d
4  5 50  e
```

```
>>> kdf.prod()
A          120
B    12000000
dtype: int64
```

If there is no numeric type columns, returns empty Series.

```
>>> ks.DataFrame({"key": ['a', 'b', 'c'], "val": ['x', 'y', 'z']}).prod()
Series([], dtype: float64)
```

On a Series:

```
>>> ks.Series([1, 2, 3, 4, 5]).prod()
120
```

By default, the product of an empty or all-NA Series is 1

```
>>> ks.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> ks.Series([]).prod(min_count=1)
nan
```

**databricks.koalas.DataFrame.quantile**

`DataFrame.quantile` (*q*: `Union[float, Iterable[float]] = 0.5`, *axis*: `Union[int, str] = 0`, *numeric\_only*: `bool = True`, *accuracy*: `int = 10000`)  $\rightarrow$  `Union[DataFrame, Series]`

Return value at the given quantile.

---

**Note:** Unlike pandas', the quantile in Koalas is an approximated quantile based upon approximate percentile computation because computing quantile across a large dataset is extremely expensive.

---

**Parameters**

**q** [float or array-like, default 0.5 (50% quantile)]  $0 \leq q \leq 1$ , the quantile(s) to compute.

**axis** [int or str, default 0 or 'index'] Can only be set to 0 at the moment.

**numeric\_only** [bool, default True] If False, the quantile of datetime and timedelta data will be computed as well. Can only be set to True at the moment.

**accuracy** [int, optional] Default accuracy of approximation. Larger value means better accuracy. The relative error can be deduced by  $1.0 / \text{accuracy}$ .

**Returns**

**Series or DataFrame** If *q* is an array, a DataFrame will be returned where the index is *q*, the columns are the columns of self, and the values are the quantiles. If *q* is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

**Examples**

```
>>> kdf = ks.DataFrame({'a': [1, 2, 3, 4, 5], 'b': [6, 7, 8, 9, 0]})
>>> kdf
   a  b
0  1  6
1  2  7
2  3  8
3  4  9
4  5  0
```

```
>>> kdf.quantile(.5)
a    3.0
b    7.0
Name: 0.5, dtype: float64
```

```
>>> kdf.quantile([.25, .5, .75])
      a    b
0.25  2.0  6.0
0.50  3.0  7.0
0.75  4.0  8.0
```



**databricks.koalas.DataFrame.nunique**

`DataFrame.nunique` (*axis*: Union[int, str] = 0, *dropna*: bool = True, *approx*: bool = False, *rsd*: float = 0.05) → Series

Return number of unique elements in the object.

Excludes NA values by default.

**Parameters**

**axis** [int, default 0 or 'index'] Can only be set to 0 at the moment.

**dropna** [bool, default True] Don't include NaN in the count.

**approx: bool, default False** If False, will use the exact algorithm and return the exact number of unique. If True, it uses the HyperLogLog approximate algorithm, which is significantly faster for large amount of data. Note: This parameter is specific to Koalas and is not found in pandas.

**rsd: float, default 0.05** Maximum estimation error allowed in the HyperLogLog algorithm. Note: Just like approx this parameter is specific to Koalas.

**Returns**

The number of unique values per column as a Koalas Series.

**Examples**

```
>>> df = ks.DataFrame({'A': [1, 2, 3], 'B': [np.nan, 3, np.nan]})
>>> df.nunique()
A      3
B      1
dtype: int64
```

```
>>> df.nunique(dropna=False)
A      3
B      2
dtype: int64
```

On big data, we recommend using the approximate algorithm to speed up this function. The result will be very close to the exact unique count.

```
>>> df.nunique(approx=True)
A      3
B      1
dtype: int64
```

**databricks.koalas.DataFrame.sem**

`DataFrame.sem` (*axis*: Union[int, str] = None, *ddof*: int = 1, *numeric\_only*: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]

Return unbiased standard error of the mean over requested axis.

**Parameters**

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**Returns**

**scalar(for Series) or Series(for DataFrame)**

**Examples**

```
>>> kdf = ks.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
>>> kdf
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> kdf.sem()
a    0.57735
b    0.57735
dtype: float64
```

```
>>> kdf.sem(ddof=0)
a    0.471405
b    0.471405
dtype: float64
```

```
>>> kdf.sem(axis=1)
0    1.5
1    1.5
2    1.5
dtype: float64
```

**Support for Series**

```
>>> kser = kdf.a
>>> kser
0    1
1    2
2    3
Name: a, dtype: int64
```

```
>>> kser.sem()
0.5773502691896258
```

```
>>> kser.sem(ddof=0)
0.47140452079103173
```

**databricks.koalas.DataFrame.skew**

`DataFrame.skew` (*axis*: `Union[int, str] = None`, *numeric\_only*: `bool = None`) → `Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]`  
 Return unbiased skew normalized by N-1.

**Parameters**

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**Returns**

**skew** [scalar for a Series, and a Series for a DataFrame.]

**Examples**

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                    columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.skew()
a      0.000000e+00
b     -3.319678e-16
dtype: float64
```

On a Series:

```
>>> df['a'].skew()
0.0
```

**databricks.koalas.DataFrame.sum**

`DataFrame.sum` (*axis*: `Union[int, str] = None`, *numeric\_only*: `bool = None`, *min\_count*: `int = 0`) → `Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]`  
 Return the sum of the values.

**Parameters**

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**min\_count** [int, default 0]

**The required number of valid values to perform the operation. If fewer than min\_count non-NA values are present the result will be NA.**

**Returns**

**sum** [scalar for a Series, and a Series for a DataFrame.]

## Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, np.nan, 0.3, np.nan]},
...                     columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.sum()
a      6.0
b      0.4
dtype: float64
```

```
>>> df.sum(axis=1)
0      1.1
1      2.0
2      3.3
3      0.0
dtype: float64
```

```
>>> df.sum(min_count=3)
a      6.0
b      NaN
dtype: float64
```

```
>>> df.sum(axis=1, min_count=1)
0      1.1
1      2.0
2      3.3
3      NaN
dtype: float64
```

On a Series:

```
>>> df['a'].sum()
6.0
```

```
>>> df['a'].sum(min_count=3)
6.0
>>> df['b'].sum(min_count=3)
nan
```

## databricks.koalas.DataFrame.std

`DataFrame.std(axis: Union[int, str] = None, ddof: int = 1, numeric_only: bool = None) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]`

Return sample standard deviation.

### Parameters

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**Returns**

**std** [scalar for a Series, and a Series for a DataFrame.]

**Examples**

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                     columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.std()
a    1.0
b    0.1
dtype: float64
```

```
>>> df.std(axis=1)
0    0.636396
1    1.272792
2    1.909188
3         NaN
dtype: float64
```

```
>>> df.std(ddof=0)
a    0.816497
b    0.081650
dtype: float64
```

On a Series:

```
>>> df['a'].std()
1.0
```

```
>>> df['a'].std(ddof=0)
0.816496580927726
```

**databricks.koalas.DataFrame.var**

**DataFrame.var** (*axis: Union[int, str] = None, ddof: int = 1, numeric\_only: bool = None*) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Series]  
Return unbiased variance.

**Parameters**

**axis** [{index (0), columns (1)}] Axis for the function to be applied on.

**ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**numeric\_only** [bool, default None] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

**Returns**

**var** [scalar for a Series, and a Series for a DataFrame.]

## Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, np.nan], 'b': [0.1, 0.2, 0.3, np.nan]},
...                     columns=['a', 'b'])
```

On a DataFrame:

```
>>> df.var()
a    1.00
b    0.01
dtype: float64
```

```
>>> df.var(axis=1)
0    0.405
1    1.620
2    3.645
3     NaN
dtype: float64
```

```
>>> df.var(ddof=0)
a    0.666667
b    0.006667
dtype: float64
```

On a Series:

```
>>> df['a'].var()
1.0
```

```
>>> df['a'].var(ddof=0)
0.6666666666666666
```

## `databricks.koalas.DataFrame.cummin`

`DataFrame.cummin` (*skipna: bool = True*) → Union[Series, DataFrame]

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

---

**Note:** the current implementation of `cummin` uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

### Parameters

**skipna** [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

### Returns

**DataFrame or Series**

See also:

**DataFrame.min** Return the minimum over DataFrame axis.

**DataFrame.cummax** Return cumulative maximum over DataFrame axis.

**DataFrame.cummin** Return cumulative minimum over DataFrame axis.

**DataFrame.cumsum** Return cumulative sum over DataFrame axis.

**Series.min** Return the minimum over Series axis.

**Series.cummax** Return cumulative maximum over Series axis.

**Series.cummin** Return cumulative minimum over Series axis.

**Series.cumsum** Return cumulative sum over Series axis.

**Series.cumprod** Return cumulative product over Series axis.

## Examples

```
>>> df = ks.DataFrame([[2.0, 1.0], [3.0, None], [1.0, 0.0]], columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column.

```
>>> df.cummin()
   A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

It works identically in Series.

```
>>> df.A.cummin()
0    2.0
1    2.0
2    1.0
Name: A, dtype: float64
```

## databricks.koalas.DataFrame.cummax

`DataFrame.cummax(skipna: bool = True) → Union[Series, DataFrame]`

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

---

**Note:** the current implementation of cummax uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

## Parameters

**skipna** [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

### Returns

**DataFrame or Series**

See also:

**DataFrame.max** Return the maximum over DataFrame axis.

**DataFrame.cummax** Return cumulative maximum over DataFrame axis.

**DataFrame.cummin** Return cumulative minimum over DataFrame axis.

**DataFrame.cumsum** Return cumulative sum over DataFrame axis.

**DataFrame.cumprod** Return cumulative product over DataFrame axis.

**Series.max** Return the maximum over Series axis.

**Series.cummax** Return cumulative maximum over Series axis.

**Series.cummin** Return cumulative minimum over Series axis.

**Series.cumsum** Return cumulative sum over Series axis.

**Series.cumprod** Return cumulative product over Series axis.

### Examples

```
>>> df = ks.DataFrame([[2.0, 1.0], [3.0, None], [1.0, 0.0]], columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column.

```
>>> df.cummax()
   A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

It works identically in Series.

```
>>> df.B.cummax()
0    1.0
1    NaN
2    1.0
Name: B, dtype: float64
```



**databricks.koalas.DataFrame.cumsum**

`DataFrame.cumsum(skipna: bool = True) → Union[Series, DataFrame]`

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

---

**Note:** the current implementation of `cumsum` uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

**Parameters**

**skipna** [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**Returns**

**DataFrame or Series**

See also:

`DataFrame.sum` Return the sum over DataFrame axis.

`DataFrame.cummax` Return cumulative maximum over DataFrame axis.

`DataFrame.cummin` Return cumulative minimum over DataFrame axis.

`DataFrame.cumsum` Return cumulative sum over DataFrame axis.

`DataFrame.cumprod` Return cumulative product over DataFrame axis.

`Series.sum` Return the sum over Series axis.

`Series.cummax` Return cumulative maximum over Series axis.

`Series.cummin` Return cumulative minimum over Series axis.

`Series.cumsum` Return cumulative sum over Series axis.

`Series.cumprod` Return cumulative product over Series axis.

**Examples**

```
>>> df = ks.DataFrame([[2.0, 1.0], [3.0, None], [1.0, 0.0]], columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column.

```
>>> df.cumsum()
   A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

It works identically in Series.

```
>>> df.A.cumsum()
0    2.0
1    5.0
2    6.0
Name: A, dtype: float64
```

### **databricks.koalas.DataFrame.cumprod**

`DataFrame.cumprod(skipna: bool = True) → Union[Series, DataFrame]`

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

---

**Note:** the current implementation of `cumprod` uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

---

**Note:** unlike pandas', Koalas' emulates cumulative product by `exp(sum(log(...)))` trick. Therefore, it only works for positive numbers.

---

#### **Parameters**

**skipna** [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

#### **Returns**

**DataFrame or Series**

#### **Raises**

**Exception** [If the values is equal to or lower than 0.]

**See also:**

**DataFrame.cummax** Return cumulative maximum over DataFrame axis.

**DataFrame.cummin** Return cumulative minimum over DataFrame axis.

**DataFrame.cumsum** Return cumulative sum over DataFrame axis.

**DataFrame.cumprod** Return cumulative product over DataFrame axis.

**Series.cummax** Return cumulative maximum over Series axis.

**Series.cummin** Return cumulative minimum over Series axis.

**Series.cumsum** Return cumulative sum over Series axis.

**Series.cumprod** Return cumulative product over Series axis.

## Examples

```
>>> df = ks.DataFrame([[2.0, 1.0], [3.0, None], [4.0, 10.0]], columns=list('AB'))
>>> df
   A      B
0  2.0    1.0
1  3.0   NaN
2  4.0   10.0
```

By default, iterates over rows and finds the sum in each column.

```
>>> df.cumprod()
   A      B
0  2.0    1.0
1  6.0   NaN
2 24.0   10.0
```

It works identically in Series.

```
>>> df.A.cumprod()
0      2.0
1      6.0
2     24.0
Name: A, dtype: float64
```

## databricks.koalas.DataFrame.round

`DataFrame.round(decimals=0) → databricks.koalas.frame.DataFrame`

Round a DataFrame to a variable number of decimal places.

### Parameters

**decimals** [int, dict, Series] Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and Series round to variable numbers of places. Column names should be in the keys if *decimals* is a dict-like, or in the index if *decimals* is a Series. Any columns not included in *decimals* will be left as is. Elements of *decimals* which are not columns of the input will be ignored.

---

**Note:** If *decimals* is a Series, it is expected to be small, as all the data is loaded into the driver's memory.

---

### Returns

**DataFrame**

See also:

[\*Series.round\*](#)

## Examples

```
>>> df = ks.DataFrame({'A': [0.028208, 0.038683, 0.877076],
...                    'B': [0.992815, 0.645646, 0.149370],
...                    'C': [0.173891, 0.577595, 0.491027]},
...                    columns=['A', 'B', 'C'],
...                    index=['first', 'second', 'third'])
>>> df
```

	A	B	C
first	0.028208	0.992815	0.173891
second	0.038683	0.645646	0.577595
third	0.877076	0.149370	0.491027

```
>>> df.round(2)
```

	A	B	C
first	0.03	0.99	0.17
second	0.04	0.65	0.58
third	0.88	0.15	0.49

```
>>> df.round({'A': 1, 'C': 2})
```

	A	B	C
first	0.0	0.992815	0.17
second	0.0	0.645646	0.58
third	0.9	0.149370	0.49

```
>>> decimals = ks.Series([1, 0, 2], index=['A', 'B', 'C'])
>>> df.round(decimals)
```

	A	B	C
first	0.0	1.0	0.17
second	0.0	1.0	0.58
third	0.9	0.0	0.49

## databricks.koalas.DataFrame.diff

`DataFrame.diff` (*periods: int = 1, axis: Union[int, str] = 0*) → `databricks.koalas.frame.DataFrame`  
First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

---

**Note:** the current implementation of `diff` uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

### Parameters

**periods** [int, default 1] Periods to shift for calculating difference, accepts negative values.

**axis** [int, default 0 or 'index'] Can only be set to 0 at the moment.

### Returns

**diffed** [DataFrame]

## Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                     'b': [1, 1, 2, 3, 5, 8],
...                     'c': [1, 4, 9, 16, 25, 36]}, columns=['a', 'b', 'c'])
>>> df
   a  b  c
0  1  1  1
1  2  1  4
2  3  2  9
3  4  3 16
4  5  5 25
5  6  8 36
```

```
>>> df.diff()
   a  b  c
0 NaN NaN NaN
1 1.0 0.0 3.0
2 1.0 1.0 5.0
3 1.0 1.0 7.0
4 1.0 2.0 9.0
5 1.0 3.0 11.0
```

Difference with previous column

```
>>> df.diff(periods=3)
   a  b  c
0 NaN NaN NaN
1 NaN NaN NaN
2 NaN NaN NaN
3 3.0 2.0 15.0
4 3.0 4.0 21.0
5 3.0 6.0 27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
   a  b  c
0 -1.0 0.0 -3.0
1 -1.0 -1.0 -5.0
2 -1.0 -1.0 -7.0
3 -1.0 -2.0 -9.0
4 -1.0 -3.0 -11.0
5 NaN NaN NaN
```

## `databricks.koalas.DataFrame.eval`

`DataFrame.eval` (*expr*, *inplace=False*) → Union[DataFrame, Series, None]

Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

### Parameters

**expr** [str] The expression string to evaluate.

**inplace** [bool, default False] If the expression contains an assignment, whether to perform the operation inplace and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

### Returns

The result of the evaluation.

See also:

**DataFrame.query** Evaluates a boolean expression to query the columns of a frame.

**DataFrame.assign** Can evaluate an expression or function to create new values for a column.

**eval** Evaluate a Python expression as a string using various backends.

### Examples

```
>>> df = ks.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
>>> df
   A  B
0  1 10
1  2  8
2  3  6
3  4  4
4  5  2
>>> df.eval('A + B')
0    11
1    10
2     9
3     8
4     7
dtype: int64
```

Assignment is allowed though by default the original DataFrame is not modified.

```
>>> df.eval('C = A + B')
   A  B  C
0  1 10 11
1  2  8 10
2  3  6  9
3  4  4  8
4  5  2  7
>>> df
   A  B
0  1 10
1  2  8
2  3  6
3  4  4
4  5  2
```

Use `inplace=True` to modify the original DataFrame.

```
>>> df.eval('C = A + B', inplace=True)
>>> df
   A  B  C
0  1 10 11
1  2  8 10
```

(continues on next page)

(continued from previous page)

2	3	6	9
3	4	4	8
4	5	2	7

### 3.4.8 Reindexing / Selection / Label manipulation

<code>DataFrame.add_prefix(prefix)</code>	Prefix labels with string <i>prefix</i> .
<code>DataFrame.add_suffix(suffix)</code>	Suffix labels with string <i>suffix</i> .
<code>DataFrame.align(other[, join, axis, copy])</code>	Align two objects on their axes with the specified join method.
<code>DataFrame.at_time(time[, asof, axis])</code>	Select values at particular time of day (e.g., 9:30AM).
<code>DataFrame.between_time(start_time, end_time)</code>	Select values between particular times of the day (e.g., 9:00-9:30 AM).
<code>DataFrame.drop([labels, axis, columns])</code>	Drop specified labels from columns.
<code>DataFrame.droplevel(level[, axis])</code>	Return DataFrame with requested index / column level(s) removed.
<code>DataFrame.drop_duplicates([subset, keep, ...])</code>	Return DataFrame with duplicate rows removed, optionally only considering certain columns.
<code>DataFrame.duplicated([subset, keep])</code>	Return boolean Series denoting duplicate rows, optionally only considering certain columns.
<code>DataFrame.equals(other)</code>	Compare if the current value is equal to the other.
<code>DataFrame.filter([items, like, regex, axis])</code>	Subset rows or columns of dataframe according to labels in the specified index.
<code>DataFrame.first(offset)</code>	Select first periods of time series data based on a date offset.
<code>DataFrame.head([n])</code>	Return the first <i>n</i> rows.
<code>DataFrame.last(offset)</code>	Select final periods of time series data based on a date offset.
<code>DataFrame.rename([mapper, index, columns, ...])</code>	Alter axes labels.
<code>DataFrame.rename_axis([mapper, index, ...])</code>	Set the name of the axis for the index or columns.
<code>DataFrame.reset_index([level, drop, ...])</code>	Reset the index, or a level of it.
<code>DataFrame.set_index(keys[, drop, append, ...])</code>	Set the DataFrame index (row labels) using one or more existing columns.
<code>DataFrame.swapaxes(i, j[, copy])</code>	Interchange axes and swap values axes appropriately.
<code>DataFrame.swaplevel([i, j, axis])</code>	Swap levels <i>i</i> and <i>j</i> in a MultiIndex on a particular axis.
<code>DataFrame.take(indices[, axis])</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>DataFrame.isin(values)</code>	Whether each element in the DataFrame is contained in values.
<code>DataFrame.sample([n, frac, replace, ...])</code>	Return a random sample of items from an axis of object.
<code>DataFrame.truncate([before, after, axis, copy])</code>	Truncate a Series or DataFrame before and after some index value.

**databricks.koalas.DataFrame.add\_prefix**

`DataFrame.add_prefix(prefix)` → `databricks.koalas.frame.DataFrame`  
Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

**Parameters**

**prefix** [str] The string to add before each label.

**Returns**

**DataFrame** New DataFrame with updated labels.

See also:

**Series.add\_prefix** Prefix row labels with string *prefix*.

**Series.add\_suffix** Suffix row labels with string *suffix*.

**DataFrame.add\_suffix** Suffix column labels with string *suffix*.

**Examples**

```
>>> df = ks.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]}, columns=['A', 'B'])
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
   col_A  col_B
0      1      3
1      2      4
2      3      5
3      4      6
```

**databricks.koalas.DataFrame.add\_suffix**

`DataFrame.add_suffix(suffix)` → `databricks.koalas.frame.DataFrame`  
Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

**Parameters**

**suffix** [str] The string to add before each label.

**Returns**

**DataFrame** New DataFrame with updated labels.

See also:

**Series.add\_prefix** Prefix row labels with string *prefix*.

**Series.add\_suffix** Suffix row labels with string *suffix*.



**DataFrame.add\_prefix** Prefix column labels with string *prefix*.

### Examples

```
>>> df = ks.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]}, columns=['A', 'B'])
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
   A_col  B_col
0      1      3
1      2      4
2      3      5
3      4      6
```

### databricks.koalas.DataFrame.align

**DataFrame.align** (*other*: Union[DataFrame, Series], *join*: str = 'outer', *axis*: Union[str, int, None] = None, *copy*: bool = True) → Tuple[DataFrame, Union[DataFrame, Series]]  
Align two objects on their axes with the specified join method.

Join method is specified for each axis Index.

#### Parameters

**other** [DataFrame or Series]

**join** [{ 'outer', 'inner', 'left', 'right' }, default 'outer']

**axis** [allowed axis of the other object, default None] Align on index (0), columns (1), or both (None).

**copy** [bool, default True] Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

#### Returns

(**left**, **right**) [(DataFrame, type of other)] Aligned objects.

### Examples

```
>>> ks.set_option("compute.ops_on_diff_frames", True)
>>> df1 = ks.DataFrame({"a": [1, 2, 3], "b": ["a", "b", "c"]}, index=[10, 20, 30])
>>> df2 = ks.DataFrame({"a": [4, 5, 6], "c": ["d", "e", "f"]}, index=[10, 11, 12])
```

Align both axis:

```
>>> aligned_l, aligned_r = df1.align(df2)
>>> aligned_l.sort_index()
   a      b      c
10 1.0      a NaN
11 NaN  None NaN
```

(continues on next page)

(continued from previous page)

```

12  NaN  None NaN
20  2.0    b NaN
30  3.0    c NaN
>>> aligned_r.sort_index()
      a    b    c
10  4.0 NaN    d
11  5.0 NaN    e
12  6.0 NaN    f
20  NaN NaN  None
30  NaN NaN  None

```

Align only axis=0 (index):

```

>>> aligned_l, aligned_r = df1.align(df2, axis=0)
>>> aligned_l.sort_index()
      a    b
10  1.0    a
11  NaN  None
12  NaN  None
20  2.0    b
30  3.0    c
>>> aligned_r.sort_index()
      a    c
10  4.0    d
11  5.0    e
12  6.0    f
20  NaN  None
30  NaN  None

```

Align only axis=1 (column):

```

>>> aligned_l, aligned_r = df1.align(df2, axis=1)
>>> aligned_l.sort_index()
      a  b  c
10  1  a NaN
20  2  b NaN
30  3  c NaN
>>> aligned_r.sort_index()
      a  b  c
10  4 NaN d
11  5 NaN e
12  6 NaN f

```

Align with the join type “inner”:

```

>>> aligned_l, aligned_r = df1.align(df2, join="inner")
>>> aligned_l.sort_index()
      a
10  1
>>> aligned_r.sort_index()
      a
10  4

```

Align with a Series:

```

>>> s = ks.Series([7, 8, 9], index=[10, 11, 12])
>>> aligned_l, aligned_r = df1.align(s, axis=0)

```

(continues on next page)

(continued from previous page)

```
>>> aligned_l.sort_index()
      a      b
10  1.0      a
11  NaN    None
12  NaN    None
20  2.0      b
30  3.0      c
>>> aligned_r.sort_index()
10      7.0
11      8.0
12      9.0
20      NaN
30      NaN
dtype: float64
```

```
>>> ks.reset_option("compute.ops_on_diff_frames")
```

### `databricks.koalas.DataFrame.at_time`

`DataFrame.at_time` (*time*: `Union[datetime.time, str]`, *asof*: `bool = False`, *axis*: `Union[int, str] = 0`) → `databricks.koalas.frame.DataFrame`  
 Select values at particular time of day (e.g., 9:30AM).

#### Parameters

**time** [datetime.time or str]  
**axis** [{0 or 'index', 1 or 'columns'}, default 0]

#### Returns

**DataFrame**

#### Raises

**TypeError** If the index is not a `DatetimeIndex`

See also:

`between_time` Select values between particular times of the day.

`DatetimeIndex.indexer_at_time` Get just the index locations for values at particular time of the day.

### Examples

```
>>> idx = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> kdf = ks.DataFrame({'A': [1, 2, 3, 4]}, index=idx)
>>> kdf
              A
2018-04-09 00:00:00  1
2018-04-09 12:00:00  2
2018-04-10 00:00:00  3
2018-04-10 12:00:00  4
```

```
>>> kdf.at_time('12:00')
      A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

### `databricks.koalas.DataFrame.between_time`

`DataFrame.between_time` (*start\_time*: *Union[datetime.time, str]*, *end\_time*: *Union[datetime.time, str]*, *include\_start*: *bool = True*, *include\_end*: *bool = True*, *axis*: *Union[int, str] = 0*) → `databricks.koalas.frame.DataFrame`

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting *start\_time* to be later than *end\_time*, you can get the times that are *not* between the two times.

#### Parameters

**start\_time** [*datetime.time* or *str*] Initial time as a time filter limit.

**end\_time** [*datetime.time* or *str*] End time as a time filter limit.

**include\_start** [*bool*, default *True*] Whether the start time needs to be included in the result.

**include\_end** [*bool*, default *True*] Whether the end time needs to be included in the result.

**axis** [{0 or 'index', 1 or 'columns'}, default 0] Determine range time on index or columns value.

#### Returns

**DataFrame** Data from the original object filtered to the specified dates range.

#### Raises

**TypeError** If the index is not a *DatetimeIndex*

#### See also:

**at\_time** Select values at a particular time of the day.

**first** Select initial periods of time series based on a date offset.

**last** Select final periods of time series based on a date offset.

**DatetimeIndex.indexer\_between\_time** Get just the index locations for values between particular times of the day.

### Examples

```
>>> idx = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> kdf = ks.DataFrame({'A': [1, 2, 3, 4]}, index=idx)
>>> kdf
      A
2018-04-09 00:00:00  1
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
2018-04-12 01:00:00  4
```

```
>>> kdf.between_time('0:15', '0:45')
      A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> kdf.between_time('0:45', '0:15')
      A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

### `databricks.koalas.DataFrame.drop`

`DataFrame.drop(labels=None, axis=1, columns: Union[Any, Tuple, List[Any], List[Tuple]] = None) →`

`databricks.koalas.DataFrame`  
Drop specified labels from columns.

Remove columns by specifying label names and `axis=1` or `columns`. When specifying both labels and columns, only labels will be dropped. Removing rows is yet to be implemented.

#### Parameters

**labels** [single label or list-like] Column labels to drop.

**axis** [{1 or 'columns'}], default 1]

**columns** [single label or list-like] Alternative to specifying axis (`labels`, `axis=1` is equivalent to `columns=labels`).

#### Returns

**dropped** [DataFrame]

See also:

[`Series.dropna`](#)

#### Notes

Currently only `axis = 1` is supported in this function, `axis = 0` is yet to be implemented.

#### Examples

```
>>> df = ks.DataFrame({'x': [1, 2], 'y': [3, 4], 'z': [5, 6], 'w': [7, 8]},
...                    columns=['x', 'y', 'z', 'w'])
>>> df
   x  y  z  w
0  1  3  5  7
1  2  4  6  8
```

```
>>> df.drop('x', axis=1)
   y  z  w
0  3  5  7
1  4  6  8
```

```
>>> df.drop(['y', 'z'], axis=1)
   x  w
0  1  7
1  2  8
```

```
>>> df.drop(columns=['y', 'z'])
   x  w
0  1  7
1  2  8
```

Also support for MultiIndex

```
>>> df = ks.DataFrame({'x': [1, 2], 'y': [3, 4], 'z': [5, 6], 'w': [7, 8]},
...                    columns=['x', 'y', 'z', 'w'])
>>> columns = [('a', 'x'), ('a', 'y'), ('b', 'z'), ('b', 'w')]
>>> df.columns = pd.MultiIndex.from_tuples(columns)
>>> df
   a      b
   x  y  z  w
0  1  3  5  7
1  2  4  6  8
>>> df.drop('a')
   b
   z  w
0  5  7
1  6  8
```

### databricks.koalas.DataFrame.droplevel

`DataFrame.droplevel` (*level*, *axis=0*) → `databricks.koalas.frame.DataFrame`

Return DataFrame with requested index / column level(s) removed.

#### Parameters

**level: int, str, or list-like** If a string is given, must be the name of a level If list-like, elements must be names or positional indexes of levels.

**axis: {0 or 'index', 1 or 'columns'}, default 0**

#### Returns

**DataFrame with requested index / column level(s) removed.**

### Examples

```
>>> df = ks.DataFrame(
...     [[3, 4], [7, 8], [11, 12]],
...     index=pd.MultiIndex.from_tuples([(1, 2), (5, 6), (9, 10)], names=["a", "b
...     ↪"]),
... )
```

```
>>> df.columns = pd.MultiIndex.from_tuples([
...     ('c', 'e'), ('d', 'f')
... ], names=['level_1', 'level_2'])
```

```
>>> df
level_1  c  d
level_2  e  f
a b
1 2      3  4
```

(continues on next page)

(continued from previous page)

```
5 6      7      8
9 10     11     12
```

```
>>> df.droplevel('a')
level_1  c    d
level_2  e    f
b
2        3    4
6        7    8
10       11   12
```

```
>>> df.droplevel('level_2', axis=1)
level_1  c    d
a b
1 2      3    4
5 6      7    8
9 10     11   12
```

### `databricks.koalas.DataFrame.drop_duplicates`

`DataFrame.drop_duplicates` (*subset=None*, *keep='first'*, *inplace=False*) → Optional[`databricks.koalas.frame.DataFrame`]  
Return `DataFrame` with duplicate rows removed, optionally only considering certain columns.

#### Parameters

**subset** [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns.

**keep** [{‘first’, ‘last’, False}, default ‘first’] Determines which duplicates (if any) to keep. - first : Drop duplicates except for the first occurrence. - last : Drop duplicates except for the last occurrence. - False : Drop all duplicates.

**inplace** [boolean, default False] Whether to drop duplicates in place or to return a copy.

#### Returns

**DataFrame** `DataFrame` with duplicates removed or None if `inplace=True`.

```
>>> df = ks.DataFrame(
    ..
```

```
... {'a': [1, 2, 2, 2, 3], 'b': ['a', 'a', 'a', 'c', 'd']}, columns = ['a', 'b'])
```

```
>>> df
a  b
```

```
0 1 a
```

```
1 2 a
```

```
2 2 a
```

```
3 2 c
```

```
4 3 d
```

```
>>> df.drop_duplicates().sort_index()  
a  b
```

**0 1 a**

**1 2 a**

**3 2 c**

**4 3 d**

```
>>> df.drop_duplicates('a').sort_index()  
a  b
```

**0 1 a**

**1 2 a**

**4 3 d**

```
>>> df.drop_duplicates(['a', 'b']).sort_index()  
a  b
```

**0 1 a**

**1 2 a**

**3 2 c**

**4 3 d**

```
>>> df.drop_duplicates(keep='last').sort_index()  
a  b
```

**0 1 a**

**2 2 a**

**3 2 c**

**4 3 d**

```
>>> df.drop_duplicates(keep=False).sort_index()  
a  b
```

**0 1 a**

**3 2 c**

**4 3 d**



**databricks.koalas.DataFrame.duplicated**

`DataFrame.duplicated` (*subset=None, keep='first'*) → Series

Return boolean Series denoting duplicate rows, optionally only considering certain columns.

**Parameters**

**subset** [column label or sequence of labels, optional] Only consider certain columns for identifying duplicates, by default use all of the columns

**keep** [{‘first’, ‘last’, False}, default ‘first’]

- `first` : Mark duplicates as True except for the first occurrence.
- `last` : Mark duplicates as True except for the last occurrence.
- `False` : Mark all duplicates as True.

**Returns**

**duplicated** [Series]

**Examples**

```
>>> df = ks.DataFrame({'a': [1, 1, 1, 3], 'b': [1, 1, 1, 4], 'c': [1, 1, 1, 5]},
...                    columns = ['a', 'b', 'c'])
>>> df
   a  b  c
0  1  1  1
1  1  1  1
2  1  1  1
3  3  4  5
```

```
>>> df.duplicated().sort_index()
0    False
1     True
2     True
3    False
dtype: bool
```

Mark duplicates as True except for the last occurrence.

```
>>> df.duplicated(keep='last').sort_index()
0     True
1     True
2    False
3    False
dtype: bool
```

Mark all duplicates as True.

```
>>> df.duplicated(keep=False).sort_index()
0     True
1     True
2     True
3    False
dtype: bool
```

**databricks.koalas.DataFrame.equals**

`DataFrame.equals` (*other*) → `databricks.koalas.frame.DataFrame`

Compare if the current value is equal to the other.

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4],
...                    'b': [1, np.nan, 1, np.nan]},
...                    index=['a', 'b', 'c', 'd'], columns=['a', 'b'])
```

```
>>> df.eq(1)
      a      b
a  True  True
b False False
c False  True
d False False
```

**databricks.koalas.DataFrame.filter**

`DataFrame.filter` (*items=None*, *like=None*, *regex=None*, *axis=None*) →

`databricks.koalas.frame.DataFrame`

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

**Parameters**

**items** [list-like] Keep labels from axis which are in items.

**like** [string] Keep labels from axis for which “like in label == True”.

**regex** [string (regular expression)] Keep labels from axis for which `re.search(regex, label) == True`.

**axis** [int or string axis name] The axis to filter on. By default this is the info axis, ‘index’ for Series, ‘columns’ for DataFrame.

**Returns**

same type as input object

See also:

[`DataFrame.loc`](#)

**Notes**

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

## Examples

```
>>> df = ks.DataFrame(np.array([[1, 2, 3], [4, 5, 6]]),
...                    index=['mouse', 'rabbit'],
...                    columns=['one', 'two', 'three'])
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
      one  three
mouse    1     3
rabbit    4     6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
      one  three
mouse    1     3
rabbit    4     6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
      one  two  three
rabbit    4    5     6
```

For a Series,

```
>>> # select rows by name
>>> df.one.filter(items=['rabbit'])
rabbit    4
Name: one, dtype: int64
```

```
>>> # select rows by regular expression
>>> df.one.filter(regex='e$')
mouse    1
Name: one, dtype: int64
```

```
>>> # select rows containing 'bbi'
>>> df.one.filter(like='bbi')
rabbit    4
Name: one, dtype: int64
```

## databricks.koalas.DataFrame.first

`DataFrame.first` (*offset*: `Union[str, pandas._libs.tslibs.offsets.DateOffset]`) → `databricks.koalas.frame.DataFrame`  
 Select first periods of time series data based on a date offset.

When having a DataFrame with dates as index, this function can select the first few rows based on a date offset.

### Parameters

**offset** [str or DateOffset] The offset length of the data that will be selected. For instance, '3D' will display all the rows having their index within the first 3 days.

### Returns

**DataFrame** A subset of the caller.

**Raises**

**TypeError** If the index is not a *DatetimeIndex*

**Examples**

```
>>> index = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> kdf = ks.DataFrame({'A': [1, 2, 3, 4]}, index=index)
>>> kdf
```

	A
2018-04-09	1
2018-04-11	2
2018-04-13	3
2018-04-15	4

Get the rows for the last 3 days:

```
>>> kdf.first('3D')
```

	A
2018-04-09	1
2018-04-11	2

Notice the data for 3 first calendar days were returned, not the first 3 observed days in the dataset, and therefore data for 2018-04-13 was not returned.

**databricks.koalas.DataFrame.last**

**DataFrame.last** (*offset*: *Union[str, pandas.\_libs.tslib.offsets.DateOffset]*) → *databricks.koalas.frame.DataFrame*  
 Select final periods of time series data based on a date offset.

When having a DataFrame with dates as index, this function can select the last few rows based on a date offset.

**Parameters**

**offset** [str or DateOffset] The offset length of the data that will be selected. For instance, '3D' will display all the rows having their index within the last 3 days.

**Returns**

**DataFrame** A subset of the caller.

**Raises**

**TypeError** If the index is not a *DatetimeIndex*

**Examples**

```
>>> index = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> kdf = ks.DataFrame({'A': [1, 2, 3, 4]}, index=index)
>>> kdf
```

	A
2018-04-09	1
2018-04-11	2
2018-04-13	3
2018-04-15	4

Get the rows for the last 3 days:

```
>>> kdf.last('3D')
      A
2018-04-13  3
2018-04-15  4
```

Notice the data for 3 last calendar days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

## databricks.koalas.DataFrame.rename

`DataFrame.rename` (*mapper=None*, *index=None*, *columns=None*, *axis='index'*, *inplace=False*, *level=None*, *errors='ignore'*) → `Optional[databricks.koalas.frame.DataFrame]`  
 Alter axes labels. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

### Parameters

**mapper** [dict-like or function] Dict-like or functions transformations to apply to that axis' values. Use either *mapper* and *axis* to specify the axis to target with *mapper*, or *index* and *columns*.

**index** [dict-like or function] Alternative to specifying axis ("mapper, axis=0" is equivalent to "index=mapper").

**columns** [dict-like or function] Alternative to specifying axis ("mapper, axis=1" is equivalent to "columns=mapper").

**axis** [int or str, default 'index'] Axis to target with mapper. Can be either the axis name ('index', 'columns') or number (0, 1).

**inplace** [bool, default False] Whether to return a new DataFrame.

**level** [int or level name, default None] In case of a MultiIndex, only rename labels in the specified level.

**errors** [{ 'ignore', 'raise' }, default 'ignore'] If 'raise', raise a *KeyError* when a dict-like *mapper*, *index*, or *columns* contains labels that are not present in the Index being transformed. If 'ignore', existing keys will be renamed and extra keys will be ignored.

### Returns

**DataFrame with the renamed axis labels.**

### Raises

**KeyError** If any of the labels is not found in the selected axis and "errors='raise'".

## Examples

```
>>> kdf1 = ks.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> kdf1.rename(columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> kdf1.rename(index={1: 10, 2: 20})
   A  B
0   1  4
10  2  5
20  3  6
```

```
>>> def str_lower(s) -> str:
...     return str.lower(s)
>>> kdf1.rename(str_lower, axis='columns')
   a  b
0   1  4
1   2  5
2   3  6
```

```
>>> def mul10(x) -> int:
...     return x * 10
>>> kdf1.rename(mul10, axis='index')
   A  B
0   1  4
10  2  5
20  3  6
```

```
>>> idx = pd.MultiIndex.from_tuples([('X', 'A'), ('X', 'B'), ('Y', 'C'), ('Y', 'D')
↳ ''])
>>> kdf2 = ks.DataFrame([[1, 2, 3, 4], [5, 6, 7, 8]], columns=idx)
>>> kdf2.rename(columns=str_lower, level=0)
      x      y
   A  B  C  D
0   1  2  3  4
1   5  6  7  8
```

```
>>> kdf3 = ks.DataFrame([[1, 2], [3, 4], [5, 6], [7, 8]], index=idx, columns=list(
↳ 'ab'))
>>> kdf3.rename(index=str_lower)
      a  b
x a   1  2
  b   3  4
y c   5  6
  d   7  8
```

### `databricks.koalas.DataFrame.rename_axis`

`DataFrame.rename_axis` (*mapper: Optional[Any] = None, index: Optional[Any] = None, columns: Optional[Any] = None, axis: Union[str, int, None] = 0, inplace: Optional[bool] = False*) → `Optional[databricks.koalas.frame.DataFrame]`

Set the name of the axis for the index or columns.

#### Parameters

**mapper** [scalar, list-like, optional] A scalar, list-like, dict-like or functions transformations to apply to the axis name attribute.

**index, columns** [scalar, list-like, dict-like or function, optional] A scalar, list-like, dict-like or functions transformations to apply to that axis' values.

Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and/or `columns`.

**axis** [[0 or 'index', 1 or 'columns'], default 0] The axis to rename.

**inplace** [bool, default False] Modifies the object directly, instead of creating a new DataFrame.

### Returns

**DataFrame**, or **None** if *inplace* is **True**.

See also:

[`Series.rename`](#) Alter Series index labels or name.

[`DataFrame.rename`](#) Alter DataFrame index labels or name.

[`Index.rename`](#) Set new names on index.

### Notes

`DataFrame.rename_axis` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

The first calling convention will only modify the names of the index and/or the names of the Index object that is the columns.

The second calling convention will modify the names of the corresponding index specified by axis.

We *highly* recommend using keyword arguments to clarify your intent.

### Examples

```
>>> df = ks.DataFrame({"num_legs": [4, 4, 2],
...                    "num_arms": [0, 0, 2]},
...                    index=["dog", "cat", "monkey"],
...                    columns=["num_legs", "num_arms"])
>>> df
```

	num_legs	num_arms
dog	4	0
cat	4	0
monkey	2	2

```
>>> df = df.rename_axis("animal").sort_index()
>>> df
```

	num_legs	num_arms
animal		
cat	4	0
dog	4	0
monkey	2	2

```
>>> df = df.rename_axis("limbs", axis="columns").sort_index()
>>> df
```

limbs	num_legs	num_arms
animal		
cat	4	0

(continues on next page)

(continued from previous page)

dog	4	0
monkey	2	2

**MultiIndex**

```
>>> index = pd.MultiIndex.from_product(['mammal'],
...                                     ['dog', 'cat', 'monkey']],
...                                     names=['type', 'name'])
>>> df = ks.DataFrame({"num_legs": [4, 4, 2],
...                     "num_arms": [0, 0, 2]},
...                     index=index,
...                     columns=["num_legs", "num_arms"])
>>> df
```

		num_legs	num_arms
type	name		
mammal	dog	4	0
	cat	4	0
	monkey	2	2

```
>>> df.rename_axis(index={'type': 'class'}).sort_index()
```

		num_legs	num_arms
class	name		
mammal	cat	4	0
	dog	4	0
	monkey	2	2

```
>>> df.rename_axis(index=str.upper).sort_index()
```

		num_legs	num_arms
TYPE	NAME		
mammal	cat	4	0
	dog	4	0
	monkey	2	2

**databricks.koalas.DataFrame.reset\_index**

`DataFrame.reset_index(level=None, drop=False, inplace=False, col_level=0, col_fill="")` → Optional[databricks.koalas.frame.DataFrame]

Reset the index, or a level of it.

For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level\_0', 'level\_1', etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default 'index' or 'level\_0' (if 'index' is already taken) will be used.

**Parameters**

**level** [int, str, tuple, or list, default None] Only remove the given levels from the index. Removes all levels by default.

**drop** [bool, default False] Do not try to insert index into dataframe columns. This resets the index to the default integer index.

**inplace** [bool, default False] Modify the DataFrame in place (do not create a new object).

**col\_level** [int or str, default 0] If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.



**col\_fill** [object, default ‘’] If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

### Returns

**DataFrame** DataFrame with the new index.

See also:

**DataFrame.set\_index** Opposite of reset\_index.

### Examples

```
>>> df = ks.DataFrame([('bird', 389.0),
...                    ('bird', 24.0),
...                    ('mammal', 80.5),
...                    ('mammal', np.nan)],
...                    index=['falcon', 'parrot', 'lion', 'monkey'],
...                    columns=('class', 'max_speed'))
>>> df
```

	class	max_speed
falcon	bird	389.0
parrot	bird	24.0
lion	mammal	80.5
monkey	mammal	NaN

When we reset the index, the old index is added as a column. Unlike pandas, Koalas does not automatically add a sequential index. The following 0, 1, 2, 3 are only there when we display the DataFrame.

```
>>> df.reset_index()
   index  class  max_speed
0  falcon   bird    389.0
1  parrot   bird    24.0
2    lion  mammal    80.5
3  monkey  mammal     NaN
```

We can use the *drop* parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
   class  max_speed
0   bird    389.0
1   bird    24.0
2  mammal    80.5
3  mammal     NaN
```

You can also use *reset\_index* with *MultiIndex*.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
...                                   ('bird', 'parrot'),
...                                   ('mammal', 'lion'),
...                                   ('mammal', 'monkey')],
...                                   names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
...                                     ('species', 'type')])
>>> df = ks.DataFrame([(389.0, 'fly'),
...                    ( 24.0, 'fly'),
...                    ( 80.5, 'run')],
```

(continues on next page)

(continued from previous page)

```

...             (np.nan, 'jump')],
...             index=index,
...             columns=columns)
>>> df

```

		speed	species
		max	type
class	name		
bird	falcon	389.0	fly
	parrot	24.0	fly
mammal	lion	80.5	run
	monkey	NaN	jump

If the index has multiple levels, we can reset a subset of them:

```

>>> df.reset_index(level='class')

```

	class	speed	species
		max	type
name			
falcon	bird	389.0	fly
parrot	bird	24.0	fly
lion	mammal	80.5	run
monkey	mammal	NaN	jump

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```

>>> df.reset_index(level='class', col_level=1)

```

		speed	species
	class	max	type
name			
falcon	bird	389.0	fly
parrot	bird	24.0	fly
lion	mammal	80.5	run
monkey	mammal	NaN	jump

When the index is inserted under another level, we can specify under which one with the parameter *col\_fill*:

```

>>> df.reset_index(level='class', col_level=1,
...                 col_fill='species')

```

		species	speed	species
	class		max	type
name				
falcon	bird	bird	389.0	fly
parrot	bird	bird	24.0	fly
lion	mammal	mammal	80.5	run
monkey	mammal	mammal	NaN	jump

If we specify a nonexistent level for *col\_fill*, it is created:

```

>>> df.reset_index(level='class', col_level=1,
...                 col_fill='genus')

```

		genus	speed	species
	class		max	type
name				
falcon	bird	bird	389.0	fly
parrot	bird	bird	24.0	fly
lion	mammal	mammal	80.5	run
monkey	mammal	mammal	NaN	jump

**databricks.koalas.DataFrame.set\_index**

`DataFrame.set_index(keys, drop=True, append=False, inplace=False)` → Optional[databricks.koalas.frame.DataFrame]

Set the DataFrame index (row labels) using one or more existing columns.

Set the DataFrame index (row labels) using one or more existing columns or arrays (of the correct length). The index can replace the existing index or expand on it.

**Parameters**

**keys** [label or array-like or list of labels/arrays] This parameter can be either a single column key, a single array of the same length as the calling DataFrame, or a list containing an arbitrary combination of column keys and arrays. Here, “array” encompasses *Series*, *Index* and `np.ndarray`.

**drop** [bool, default True] Delete columns to be used as the new index.

**append** [bool, default False] Whether to append columns to existing index.

**inplace** [bool, default False] Modify the DataFrame in place (do not create a new object).

**Returns**

**DataFrame** Changed row labels.

See also:

**DataFrame.reset\_index** Opposite of `set_index`.

**Examples**

```
>>> df = ks.DataFrame({'month': [1, 4, 7, 10],
...                    'year': [2012, 2014, 2013, 2014],
...                    'sale': [55, 40, 84, 31]},
...                   columns=['month', 'year', 'sale'])
>>> df
   month  year  sale
0      1  2012    55
1      4  2014    40
2      7  2013    84
3     10  2014    31
```

Set the index to become the ‘month’ column:

```
>>> df.set_index('month')
      year  sale
month
1      2012    55
4      2014    40
7      2013    84
10     2014    31
```

Create a MultiIndex using columns ‘year’ and ‘month’:

```
>>> df.set_index(['year', 'month'])
      sale
year  month
2012   1      55
```

(continues on next page)

(continued from previous page)

2014	4	40
2013	7	84
2014	10	31

**databricks.koalas.DataFrame.swapaxes**

`DataFrame.swapaxes` (*i*: Union[str, int], *j*: Union[str, int], *copy*: bool = True) → databricks.koalas.frame.DataFrame  
Interchange axes and swap values axes appropriately.

**Note:** This method is based on an expensive operation due to the nature of big data. Internally it needs to generate each row for each value, and then group twice - it is a huge operation. To prevent misuse, this method has the 'compute.max\_rows' default limit of input length, and raises a ValueError.

```
>>> from databricks.koalas.config import option_context
>>> with option_context('compute.max_rows', 1000):
...     ks.DataFrame({'a': range(1001)}).swapaxes(i=0, j=1)
Traceback (most recent call last):
...
ValueError: Current DataFrame has more then the given limit 1000 rows.
Please set 'compute.max_rows' by using 'databricks.koalas.config.set_option'
to retrieve to retrieve more than 1000 rows. Note that, before changing the
'compute.max_rows', this operation is considerably expensive.
```

**Parameters**

**i:** {0 or 'index', 1 or 'columns'}. The axis to swap.

**j:** {0 or 'index', 1 or 'columns'}. The axis to swap.

**copy** [bool, default True.]

**Returns**

**DataFrame**

**Examples**

```
>>> kdf = ks.DataFrame(
...     [[1, 2, 3], [4, 5, 6], [7, 8, 9]], index=['x', 'y', 'z'], columns=['a', 'b', 'c']
... )
>>> kdf
   a  b  c
x  1  2  3
y  4  5  6
z  7  8  9
>>> kdf.swapaxes(i=1, j=0)
   x  y  z
a  1  4  7
b  2  5  8
c  3  6  9
>>> kdf.swapaxes(i=1, j=1)
```

(continues on next page)

(continued from previous page)

	a	b	c
x	1	2	3
y	4	5	6
z	7	8	9

**databricks.koalas.DataFrame.swaplevel**

`DataFrame.swaplevel(i=-2, j=-1, axis=0) → databricks.koalas.frame.DataFrame`

Swap levels i and j in a MultiIndex on a particular axis.

**Parameters**

**i, j** [int or str] Levels of the indices to be swapped. Can pass level name as string.

**axis** [{0 or 'index', 1 or 'columns'}, default 0] The axis to swap levels on. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.

**Returns**

**DataFrame** DataFrame with levels swapped in MultiIndex.

**Examples**

```
>>> midx = pd.MultiIndex.from_arrays(
...     [['red', 'blue'], [1, 2], ['s', 'm']], names = ['color', 'number', 'size
...     ↪'])
>>> midx
MultiIndex([( 'red', 1, 's'),
             ('blue', 2, 'm')],
            names=['color', 'number', 'size'])
```

Swap levels in a MultiIndex on index.

```
>>> kdf = ks.DataFrame({'x': [5, 6], 'y': [5, 6]}, index=midx)
>>> kdf
```

			x	y
color	number	size		
red	1	s	5	5
blue	2	m	6	6

```
>>> kdf.swaplevel()

color size number      x      y
red    s      1      5      5
blue   m      2      6      6
```

```
>>> kdf.swaplevel(0, 1)

number color size      x      y
1      red    s      5      5
2     blue   m      6      6
```

```
>>> kdf.swaplevel('number', 'size')
           x  y
color size number
red   s    1    5  5
blue  m    2    6  6
```

Swap levels in a MultiIndex on columns.

```
>>> kdf = ks.DataFrame({'x': [5, 6], 'y':[5, 6]})
>>> kdf.columns = midx
>>> kdf
color  red blue
number  1    2
size    s    m
0       5    5
1       6    6
```

```
>>> kdf.swaplevel(axis=1)
color  red blue
size    s    m
number  1    2
0       5    5
1       6    6
```

```
>>> kdf.swaplevel(axis=1)
color  red blue
size    s    m
number  1    2
0       5    5
1       6    6
```

```
>>> kdf.swaplevel(0, 1, axis=1)
number  1    2
color  red blue
size    s    m
0       5    5
1       6    6
```

```
>>> kdf.swaplevel('number', 'color', axis=1)
number  1    2
color  red blue
size    s    m
0       5    5
1       6    6
```

**databricks.koalas.DataFrame.take**

`DataFrame.take(indices, axis=0, **kwargs) → databricks.koalas.frame.DataFrame`

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

**Parameters**

**indices** [array-like] An array of ints indicating which positions to take.

**axis** [{0 or 'index', 1 or 'columns', None}, default 0] The axis on which to select elements. 0 means that we are selecting rows, 1 means that we are selecting columns.

**\*\*kwargs** For compatibility with `numpy.take()`. Has no effect on the output.

**Returns**

**taken** [same type as caller] An array-like containing the elements taken from the object.

See also:

**DataFrame.loc** Select a subset of a DataFrame by labels.

**DataFrame.iloc** Select a subset of a DataFrame by positions.

**numpy.take** Take elements from an array along an axis.

**Examples**

```
>>> df = ks.DataFrame([('falcon', 'bird', 389.0),
...                     ('parrot', 'bird', 24.0),
...                     ('lion', 'mammal', 80.5),
...                     ('monkey', 'mammal', np.nan)],
...                     columns=['name', 'class', 'max_speed'],
...                     index=[0, 2, 3, 1])
>>> df
   name  class  max_speed
0  falcon   bird    389.0
2  parrot   bird     24.0
3   lion  mammal     80.5
1  monkey  mammal      NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3]).sort_index()
   name  class  max_speed
0  falcon   bird    389.0
1  monkey  mammal      NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
   class  max_speed
0   bird    389.0
2   bird     24.0
```

(continues on next page)

(continued from previous page)

3	mammal	80.5
1	mammal	NaN

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2]).sort_index()
      name  class  max_speed
1  monkey  mammal         NaN
3    lion  mammal        80.5
```

## databricks.koalas.DataFrame.isin

`DataFrame.isin(values)` → `databricks.koalas.frame.DataFrame`

Whether each element in the DataFrame is contained in values.

### Parameters

**values** [iterable or dict] The sequence of values to test. If values is a dict, the keys must be the column names, which must match. Series and DataFrame are not supported.

### Returns

**DataFrame** DataFrame of booleans showing whether each element in the DataFrame is contained in values.

## Examples

```
>>> df = ks.DataFrame({'num_legs': [2, 4], 'num_wings': [2, 0]},
...                    index=['falcon', 'dog'],
...                    columns=['num_legs', 'num_wings'])
>>> df
      num_legs  num_wings
falcon         2         2
dog            4         0
```

When values is a list check whether every value in the DataFrame is present in the list (which animals have 0 or 2 legs or wings)

```
>>> df.isin([0, 2])
      num_legs  num_wings
falcon      True       True
dog        False       True
```

When values is a dict, we can pass values to check for each column separately:

```
>>> df.isin({'num_wings': [0, 3]})
      num_legs  num_wings
falcon      False      False
dog         False       True
```



**databricks.koalas.DataFrame.sample**

`DataFrame.sample` (*n*: *Optional[int] = None*, *frac*: *Optional[float] = None*, *replace*: *bool = False*, *random\_state*: *Optional[int] = None*) → `databricks.koalas.frame.DataFrame`

Return a random sample of items from an axis of object.

Please call this function using named argument by specifying the `frac` argument.

You can use `random_state` for reproducibility. However, note that different from pandas, specifying a seed in Koalas/Spark does not guarantee the sampled rows will be fixed. The result set depends on not only the seed, but also how the data is distributed across machines and to some extent network randomness when shuffle operations are involved. Even in the simplest case, the result set will depend on the system's CPU core count.

**Parameters**

**n** [int, optional] Number of items to return. This is currently NOT supported. Use `frac` instead.

**frac** [float, optional] Fraction of axis items to return.

**replace** [bool, default False] Sample with or without replacement.

**random\_state** [int, optional] Seed for the random number generator (if int).

**Returns**

**Series or DataFrame** A new object of same type as caller containing the sampled items.

**Examples**

```
>>> df = ks.DataFrame({'num_legs': [2, 4, 8, 0],
...                    'num_wings': [2, 0, 0, 0],
...                    'num_specimen_seen': [10, 2, 1, 8]},
...                    index=['falcon', 'dog', 'spider', 'fish'],
...                    columns=['num_legs', 'num_wings', 'num_specimen_seen'])
>>> df
```

	num_legs	num_wings	num_specimen_seen
falcon	2	2	10
dog	4	0	2
spider	8	0	1
fish	0	0	8

A random 25% sample of the `DataFrame`. Note that we use `random_state` to ensure the reproducibility of the examples.

```
>>> df.sample(frac=0.25, random_state=1)
```

	num_legs	num_wings	num_specimen_seen
falcon	2	2	10
fish	0	0	8

Extract 25% random elements from the `Series df['num_legs']`, with replacement, so the same items could appear more than once.

```
>>> df['num_legs'].sample(frac=0.4, replace=True, random_state=1)
```

falcon	2
spider	8
spider	8

Name: num\_legs, dtype: int64

Specifying the exact number of items to return is not supported at the moment.

```
>>> df.sample(n=5)
Traceback (most recent call last):
...
NotImplementedError: Function sample currently does not support specifying ...
```

## `databricks.koalas.DataFrame.truncate`

`DataFrame.truncate` (*before=None, after=None, axis=None, copy=True*) → Union[DataFrame, Series]  
Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

---

**Note:** This API is dependent on `Index.is_monotonic_increasing()` which can be expensive.

---

### Parameters

**before** [date, str, int] Truncate all rows before this index value.

**after** [date, str, int] Truncate all rows after this index value.

**axis** [{0 or 'index', 1 or 'columns'}, optional] Axis to truncate. Truncates the index (rows) by default.

**copy** [bool, default is True,] Return a copy of the truncated section.

### Returns

**type of caller** The truncated Series or DataFrame.

See also:

`DataFrame.loc` Select a subset of a DataFrame by label.

`DataFrame.iloc` Select a subset of a DataFrame by position.

## Examples

```
>>> df = ks.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

A Series has index that sorted integers.

```
>>> s = ks.Series([10, 20, 30, 40, 50, 60, 70],
...               index=[1, 2, 3, 4, 5, 6, 7])
>>> s
1    10
2    20
3    30
4    40
5    50
6    60
7    70
dtype: int64
```

```
>>> s.truncate(2, 5)
2    20
3    30
4    40
5    50
dtype: int64
```

A Series has index that sorted strings.

```
>>> s = ks.Series([10, 20, 30, 40, 50, 60, 70],
...               index=['a', 'b', 'c', 'd', 'e', 'f', 'g'])
>>> s
a    10
b    20
c    30
d    40
e    50
f    60
g    70
dtype: int64
```

```
>>> s.truncate('b', 'e')
b    20
c    30
d    40
```

(continues on next page)

(continued from previous page)

```
e      50
dtype: int64
```

### 3.4.9 Missing data handling

<code>DataFrame.backfill([axis, inplace, limit])</code>	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='bfill'</code> .
<code>DataFrame.dropna([axis, how, thresh, ...])</code>	Remove missing values.
<code>DataFrame.fillna([value, method, axis, ...])</code>	Fill NA/NaN values.
<code>DataFrame.replace([to_replace, value, ...])</code>	Returns a new DataFrame replacing a value with another value.
<code>DataFrame.bfill([axis, inplace, limit])</code>	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='bfill'</code> .
<code>DataFrame.ffill([axis, inplace, limit])</code>	Synonym for <code>DataFrame.fillna()</code> or <code>Series.fillna()</code> with <code>method='ffill'</code> .

#### `databricks.koalas.DataFrame.backfill`

`DataFrame.backfill` (*axis=None, inplace=False, limit=None*) → Union[DataFrame, Series]

Synonym for `DataFrame.fillna()` or `Series.fillna()` with `method='bfill'`.

**Note:** the current implementation of 'bfill' uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

#### Parameters

**axis** [{0 or *index*}] 1 and *columns* are not supported.

**inplace** [boolean, default False] Fill in place (do not create a new object)

**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

#### Returns

**DataFrame or Series** DataFrame or Series with NA entries filled.

## Examples

```
>>> kdf = ks.DataFrame({
...     'A': [None, 3, None, None],
...     'B': [2, 4, None, 3],
...     'C': [None, None, None, 1],
...     'D': [0, 1, 5, 4]
...     },
...     columns=['A', 'B', 'C', 'D'])
>>> kdf
```

	A	B	C	D
0	NaN	2.0	NaN	0
1	3.0	4.0	NaN	1
2	NaN	NaN	NaN	5
3	NaN	3.0	1.0	4

Propagate non-null values backward.

```
>>> kdf.bfill()
```

	A	B	C	D
0	3.0	2.0	1.0	0
1	3.0	4.0	1.0	1
2	NaN	3.0	1.0	5
3	NaN	3.0	1.0	4

### For Series

```
>>> kser = ks.Series([None, None, None, 1])
>>> kser
```

0	NaN
1	NaN
2	NaN
3	1.0

dtype: float64

```
>>> kser.bfill()
```

0	1.0
1	1.0
2	1.0
3	1.0

dtype: float64

## databricks.koalas.DataFrame.dropna

`DataFrame.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)` → Optional[databricks.koalas.frame.DataFrame]  
Remove missing values.

### Parameters

**axis** [{0 or 'index'}], default 0] Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.

**how** [{ 'any', 'all' }, default 'any'] Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- ‘any’ : If any NA values are present, drop that row or column.
- ‘all’ : If all values are NA, drop that row or column.

**thresh** [int, optional] Require that many non-NA values.

**subset** [array-like, optional] Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

**inplace** [bool, default False] If True, do operation inplace and return None.

### Returns

**DataFrame** DataFrame with NA entries dropped from it.

See also:

**DataFrame.drop** Drop specified labels from columns.

**DataFrame.isnull** Indicate missing values.

**DataFrame.notnull** Indicate existing (non-missing) values.

### Examples

```
>>> df = ks.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [None, 'Batmobile', 'Bullwhip'],
...                    "born": [None, "1940-04-25", None]},
...                    columns=['name', 'toy', 'born'])
>>> df
```

	name	toy	born
0	Alfred	None	None
1	Batman	Batmobile	1940-04-25
2	Catwoman	Bullwhip	None

Drop the rows where at least one element is missing.

```
>>> df.dropna()
```

	name	toy	born
1	Batman	Batmobile	1940-04-25

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
```

	name
0	Alfred
1	Batman
2	Catwoman

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
```

	name	toy	born
0	Alfred	None	None
1	Batman	Batmobile	1940-04-25
2	Catwoman	Bullwhip	None

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
   name      toy      born
1  Batman  Batmobile  1940-04-25
2  Catwoman  Bullwhip      None
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'born'])
   name      toy      born
1  Batman  Batmobile  1940-04-25
```

Keep the DataFrame with valid entries in the same variable.

```
>>> df.dropna(inplace=True)
>>> df
   name      toy      born
1  Batman  Batmobile  1940-04-25
```

### `databricks.koalas.DataFrame.fillna`

`DataFrame.fillna` (*value=None, method=None, axis=None, inplace=False, limit=None*) → Optional[databricks.koalas.frame.DataFrame]  
Fill NA/NaN values.

**Note:** the current implementation of ‘method’ parameter in `fillna` uses Spark’s Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

#### Parameters

- value** [scalar, dict, Series] Value to use to fill holes. alternately a dict/Series of values specifying which value to use for each column. DataFrame is not supported.
- method** [{‘backfill’, ‘bfill’, ‘pad’, ‘ffill’, None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap
- axis** [{0 or index}] 1 and *columns* are not supported.
- inplace** [boolean, default False] Fill in place (do not create a new object)
- limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

#### Returns

**DataFrame** DataFrame with NA entries filled.

## Examples

```
>>> df = ks.DataFrame({
...     'A': [None, 3, None, None],
...     'B': [2, 4, None, 3],
...     'C': [None, None, None, 1],
...     'D': [0, 1, 5, 4]
... },
... columns=['A', 'B', 'C', 'D'])
>>> df
```

	A	B	C	D
0	NaN	2.0	NaN	0
1	3.0	4.0	NaN	1
2	NaN	NaN	NaN	5
3	NaN	3.0	1.0	4

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
```

	A	B	C	D
0	0.0	2.0	0.0	0
1	3.0	4.0	0.0	1
2	0.0	0.0	0.0	5
3	0.0	3.0	1.0	4

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
```

	A	B	C	D
0	NaN	2.0	NaN	0
1	3.0	4.0	NaN	1
2	3.0	4.0	NaN	5
3	3.0	3.0	1.0	4

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
```

	A	B	C	D
0	0.0	2.0	2.0	0
1	3.0	4.0	2.0	1
2	0.0	1.0	2.0	5
3	0.0	3.0	1.0	4

## databricks.koalas.DataFrame.replace

`DataFrame.replace(to_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad') → Optional[databricks.koalas.frame.DataFrame]`

Returns a new DataFrame replacing a value with another value.

### Parameters

**to\_replace** [int, float, string, list, tuple or dict] Value to be replaced.

**value** [int, float, string, list or tuple] Value to use to replace holes. The replacement value must be an int, float, or string. If value is a list or tuple, value should be of the same length with to\_replace.



**inplace** [boolean, default False] Fill in place (do not create a new object)

### Returns

**DataFrame** Object after replacement.

### Examples

```
>>> df = ks.DataFrame({"name": ['Ironman', 'Captain America', 'Thor', 'Hulk'],
...                    "weapon": ['Mark-45', 'Shield', 'Mjolnir', 'Smash']},
...                    columns=['name', 'weapon'])
>>> df
```

	name	weapon
0	Ironman	Mark-45
1	Captain America	Shield
2	Thor	Mjolnir
3	Hulk	Smash

Scalar *to\_replace* and *value*

```
>>> df.replace('Ironman', 'War-Machine')
      name  weapon
0  War-Machine  Mark-45
1  Captain America  Shield
2      Thor  Mjolnir
3      Hulk  Smash
```

List like *to\_replace* and *value*

```
>>> df.replace(['Ironman', 'Captain America'], ['Rescue', 'Hawkeye'],
...             inplace=True)
>>> df
```

	name	weapon
0	Rescue	Mark-45
1	Hawkeye	Shield
2	Thor	Mjolnir
3	Hulk	Smash

Dicts can be used to specify different replacement values for different existing values To use a dict in this way the value parameter should be None

```
>>> df.replace({'Mjolnir': 'Stormbuster'})
      name  weapon
0  Rescue  Mark-45
1  Hawkeye  Shield
2      Thor  Stormbuster
3      Hulk  Smash
```

Dict can specify that different values should be replaced in different columns The value parameter should not be None in this case

```
>>> df.replace({'weapon': 'Mjolnir'}, 'Stormbuster')
      name  weapon
0  Rescue  Mark-45
1  Hawkeye  Shield
2      Thor  Stormbuster
3      Hulk  Smash
```

Nested dictionaries The value parameter should be None to use a nested dict in this way

```
>>> df.replace({'weapon': {'Mjolnir': 'Stormbuster'}})
   name      weapon
0  Rescue    Mark-45
1 Hawkeye    Shield
2   Thor  Stormbuster
3   Hulk      Smash
```

### databricks.koalas.DataFrame.bfill

DataFrame.**bfill** (*axis=None, inplace=False, limit=None*) → Union[DataFrame, Series]

Synonym for *DataFrame.fillna()* or *Series.fillna()* with *method='bfill'*.

**Note:** the current implementation of ‘bfill’ uses Spark’s Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

#### Parameters

**axis** [{0 or *index*}] 1 and *columns* are not supported.

**inplace** [boolean, default False] Fill in place (do not create a new object)

**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

#### Returns

**DataFrame or Series** DataFrame or Series with NA entries filled.

### Examples

```
>>> kdf = ks.DataFrame({
...     'A': [None, 3, None, None],
...     'B': [2, 4, None, 3],
...     'C': [None, None, None, 1],
...     'D': [0, 1, 5, 4]
... },
... columns=['A', 'B', 'C', 'D'])
>>> kdf
   A    B    C    D
0 NaN  2.0 NaN  0
1 3.0  4.0 NaN  1
2 NaN  NaN  NaN  5
3 NaN  3.0  1.0  4
```

Propagate non-null values backward.

```
>>> kdf.bfill()
   A    B    C  D
0  3.0  2.0  1.0  0
1  3.0  4.0  1.0  1
2  NaN  3.0  1.0  5
3  NaN  3.0  1.0  4
```

#### For Series

```
>>> kser = ks.Series([None, None, None, 1])
>>> kser
0    NaN
1    NaN
2    NaN
3     1.0
dtype: float64
```

```
>>> kser.bfill()
0     1.0
1     1.0
2     1.0
3     1.0
dtype: float64
```

### `databricks.koalas.DataFrame.fill`

`DataFrame.fill(axis=None, inplace=False, limit=None) → Union[DataFrame, Series]`

Synonym for `DataFrame.fillna()` or `Series.fillna()` with `method='ffill'`.

---

**Note:** the current implementation of 'fill' uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

#### Parameters

**axis** [{0 or index}] 1 and *columns* are not supported.

**inplace** [boolean, default False] Fill in place (do not create a new object)

**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

#### Returns

**DataFrame or Series** DataFrame or Series with NA entries filled.

## Examples

```
>>> kdf = ks.DataFrame({
...     'A': [None, 3, None, None],
...     'B': [2, 4, None, 3],
...     'C': [None, None, None, 1],
...     'D': [0, 1, 5, 4]
...     },
...     columns=['A', 'B', 'C', 'D'])
>>> kdf
```

	A	B	C	D
0	NaN	2.0	NaN	0
1	3.0	4.0	NaN	1
2	NaN	NaN	NaN	5
3	NaN	3.0	1.0	4

Propagate non-null values forward.

```
>>> kdf.ffmpeg()

   A    B    C  D
0 NaN  2.0 NaN  0
1 3.0  4.0 NaN  1
2 3.0  4.0 NaN  5
3 3.0  3.0 1.0  4
```

### For Series

```
>>> kser = ks.Series([2, 4, None, 3])
>>> kser
```

0	2.0
1	4.0
2	NaN
3	3.0

dtype: float64

```
>>> kser.ffmpeg()

   0    2.0
   1    4.0
   2    4.0
   3    3.0
dtype: float64
```

## 3.4.10 Reshaping, sorting, transposing

<code>DataFrame.pivot_table([values, index, ...])</code>	Create a spreadsheet-style pivot table as a DataFrame.
<code>DataFrame.pivot([index, columns, values])</code>	Return reshaped DataFrame organized by given index / column values.
<code>DataFrame.sort_index([axis, level, ...])</code>	Sort object by labels (along an axis)
<code>DataFrame.sort_values(by[, ascending, ...])</code>	Sort by the values along either axis.
<code>DataFrame.nlargest(n, columns)</code>	Return the first <i>n</i> rows ordered by <i>columns</i> in descending order.
<code>DataFrame.nsmallest(n, columns)</code>	Return the first <i>n</i> rows ordered by <i>columns</i> in ascending order.

continues on next page

Table 51 – continued from previous page

<code>DataFrame.stack()</code>	Stack the prescribed level(s) from columns to index.
<code>DataFrame.unstack()</code>	Pivot the (necessarily hierarchical) index labels.
<code>DataFrame.melt([id_vars, value_vars, ...])</code>	Unpivot a DataFrame from wide format to long format, optionally leaving identifier variables set.
<code>DataFrame.explode(column)</code>	Transform each element of a list-like to a row, replicating index values.
<code>DataFrame.squeeze([axis])</code>	Squeeze 1 dimensional axis objects into scalars.
<code>DataFrame.T</code>	Transpose index and columns.
<code>DataFrame.transpose()</code>	Transpose index and columns.
<code>DataFrame.reindex([labels, index, columns, ...])</code>	Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
<code>DataFrame.reindex_like(other[, copy])</code>	Return a DataFrame with matching indices as other object.
<code>DataFrame.rank([method, ascending])</code>	Compute numerical data ranks (1 through n) along axis.

**databricks.koalas.DataFrame.pivot\_table**

`DataFrame.pivot_table` (*values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill\_value=None*) → `databricks.koalas.frame.DataFrame`

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

**Parameters**

**values** [column to aggregate.] They should be either a list less than three or a string.

**index** [column (string) or list of columns] If an array is passed, it must be the same length as the data. The list should contain string.

**columns** [column] Columns used in the pivot operation. Only one column is supported and it should be a string.

**aggfunc** [function (string), dict, default mean] If dict is passed, the key is column to aggregate and value is function or list of functions.

**fill\_value** [scalar, default None] Value to replace missing values with.

**Returns**

**table** [DataFrame]

**Examples**

```
>>> df = ks.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                           "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                           "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                           "small", "large", "small", "small",
...                           "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
...                    "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]},
...                    columns=['A', 'B', 'C', 'D', 'E'])
>>> df
```

(continues on next page)

(continued from previous page)

	A	B	C	D	E
0	foo	one	small	1	2
1	foo	one	large	2	4
2	foo	one	large	2	5
3	foo	two	small	3	5
4	foo	two	small	3	6
5	bar	one	large	4	6
6	bar	one	small	5	8
7	bar	two	small	6	9
8	bar	two	large	7	9

This first example aggregates values by taking the sum.

```
>>> table = df.pivot_table(values='D', index=['A', 'B'],
...                          columns='C', aggfunc='sum')
>>> table.sort_index()
C      large  small
A  B
bar one    4.0    5
    two    7.0    6
foo one    4.0    1
    two    NaN    6
```

We can also fill missing values using the *fill\_value* parameter.

```
>>> table = df.pivot_table(values='D', index=['A', 'B'],
...                          columns='C', aggfunc='sum', fill_value=0)
>>> table.sort_index()
C      large  small
A  B
bar one     4     5
    two     7     6
foo one     4     1
    two     0     6
```

We can also calculate multiple types of aggregations for any given value column.

```
>>> table = df.pivot_table(values=['D'], index=['C'],
...                          columns="A", aggfunc={'D': 'mean'})
>>> table.sort_index()
D
A      bar      foo
C
large  5.5  2.000000
small  5.5  2.333333
```

The next example aggregates on multiple values.

```
>>> table = df.pivot_table(index=['C'], columns="A", values=['D', 'E'],
...                          aggfunc={'D': 'mean', 'E': 'sum'})
>>> table.sort_index()
D      E
A      bar      foo bar foo
C
large  5.5  2.000000  15    9
small  5.5  2.333333  17   13
```

**databricks.koalas.DataFrame.pivot**

`DataFrame.pivot` (*index=None, columns=None, values=None*) → `databricks.koalas.frame.DataFrame`

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a “pivot” table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation.

**Parameters**

**index** [string, optional] Column to use to make new frame’s index. If None, uses existing index.

**columns** [string] Column to use to make new frame’s columns.

**values** [string, object or a list of the previous] Column(s) to use for populating new frame’s values.

**Returns**

**DataFrame** Returns reshaped DataFrame.

See also:

**DataFrame.pivot\_table** Generalization of pivot that can handle duplicate values for one index/column pair.

**Examples**

```
>>> df = ks.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                             'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
...                    'zoo': ['x', 'y', 'z', 'q', 'w', 't']},
...                    columns=['foo', 'bar', 'baz', 'zoo'])
>>> df
   foo bar  baz zoo
0  one  A    1   x
1  one  B    2   y
2  one  C    3   z
3  two  A    4   q
4  two  B    5   w
5  two  C    6   t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz').sort_index()
...
bar  A  B  C
foo
one  1  2  3
two  4  5  6
```

```
>>> df.pivot(columns='bar', values='baz').sort_index()
bar  A    B    C
0  1.0  NaN  NaN
1  NaN  2.0  NaN
2  NaN  NaN  3.0
3  4.0  NaN  NaN
4  NaN  5.0  NaN
5  NaN  NaN  6.0
```

Notice that, unlike pandas raises an `ValueError` when duplicated values are found, Koalas' pivot still works with its first value it meets during operation because pivot is an expensive operation and it is preferred to permissively execute over failing fast when processing large data.

```
>>> df = ks.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                     "bar": ['A', 'A', 'B', 'C'],
...                     "baz": [1, 2, 3, 4]}, columns=['foo', 'bar', 'baz'])
>>> df
   foo bar baz
0  one  A   1
1  one  A   2
2  two  B   3
3  two  C   4
```

```
>>> df.pivot(index='foo', columns='bar', values='baz').sort_index()
...
bar    A    B    C
foo
one  1.0 NaN NaN
two  NaN 3.0 4.0
```

It also support multi-index and multi-index column. `>>> df.columns = pd.MultiIndex.from_tuples([('a', 'foo'), ('a', 'bar'), ('b', 'baz')])`

```
>>> df = df.set_index(('a', 'bar'), append=True)
>>> df
           a  b
      foo baz
(a, bar)
0 A      one  1
1 A      one  2
2 B      two  3
3 C      two  4
```

```
>>> df.pivot(columns=('a', 'foo'), values=('b', 'baz')).sort_index()
...
(a, 'foo')  one  two
(a, bar)
0 A          1.0 NaN
1 A          2.0 NaN
2 B          NaN 3.0
3 C          NaN 4.0
```

### `databricks.koalas.DataFrame.sort_index`

`DataFrame.sort_index` (*axis*: `int = 0`, *level*: `Union[int, List[int], None] = None`, *ascending*: `bool = True`, *inplace*: `bool = False`, *kind*: `str = None`, *na\_position*: `str = 'last'`) → `Optional[databricks.koalas.frame.DataFrame]`  
Sort object by labels (along an axis)

#### Parameters

**axis** [index, columns to direct sorting. Currently, only `axis = 0` is supported.]

**level** [int or level name or list of ints or list of level names] if not `None`, sort on values in specified index level(s)

**ascending** [boolean, default `True`] Sort ascending vs. descending



**inplace** [bool, default False] if True, perform operation in-place

**kind** [str, default None] Koalas does not allow specifying the sorting algorithm at the moment, default None

**na\_position** [{‘first’, ‘last’}, default ‘last’] first puts NaNs at the beginning, last puts NaNs at the end. Not implemented for MultiIndex.

### Returns

**sorted\_obj** [DataFrame]

### Examples

```
>>> df = ks.DataFrame({'A': [2, 1, np.nan]}, index=['b', 'a', np.nan])
```

```
>>> df.sort_index()
      A
a    1.0
b    2.0
NaN  NaN
```

```
>>> df.sort_index(ascending=False)
      A
b    2.0
a    1.0
NaN  NaN
```

```
>>> df.sort_index(na_position='first')
      A
NaN  NaN
a    1.0
b    2.0
```

```
>>> df.sort_index(inplace=True)
>>> df
      A
a    1.0
b    2.0
NaN  NaN
```

```
>>> df = ks.DataFrame({'A': range(4), 'B': range(4)[::-1]},
...                     index=[['b', 'b', 'a', 'a'], [1, 0, 1, 0]],
...                     columns=['A', 'B'])
```

```
>>> df.sort_index()
      A  B
a 0    3  0
   1    2  1
b 0    1  2
   1    0  3
```

```
>>> df.sort_index(level=1)
      A  B
a 0    3  0
```

(continues on next page)

(continued from previous page)

```
b 0 1 2
a 1 2 1
b 1 0 3
```

```
>>> df.sort_index(level=[1, 0])
      A  B
a 0  3  0
b 0  1  2
a 1  2  1
b 1  0  3
```

### databricks.koalas.DataFrame.sort\_values

`DataFrame.sort_values` (*by*: Union[Any, List[Any], Tuple, List[Tuple]], *ascending*: Union[bool, List[bool]] = True, *inplace*: bool = False, *na\_position*: str = 'last') → Optional[databricks.koalas.frame.DataFrame]

Sort by the values along either axis.

#### Parameters

**by** [str or list of str]

**ascending** [bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

**inplace** [bool, default False] if True, perform operation in-place

**na\_position** [{‘first’, ‘last’}, default ‘last’] *first* puts NaNs at the beginning, *last* puts NaNs at the end

#### Returns

**sorted\_obj** [DataFrame]

### Examples

```
>>> df = ks.DataFrame({
...     'col1': ['A', 'B', None, 'D', 'C'],
...     'col2': [2, 9, 8, 7, 4],
...     'col3': [0, 9, 4, 2, 3],
... },
...     columns=['col1', 'col2', 'col3'])
>>> df
   col1  col2  col3
0     A      2     0
1     B      9     9
2  None      8     4
3     D      7     2
4     C      4     3
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
   col1  col2  col3
0     A      2     0
```

(continues on next page)

(continued from previous page)

1	B	9	9
4	C	4	3
3	D	7	2
2	None	8	4

**Sort Descending**

```
>>> df.sort_values(by='col1', ascending=False)
   col1  col2  col3
3     D     7     2
4     C     4     3
1     B     9     9
0     A     2     0
2  None     8     4
```

**Sort by multiple columns**

```
>>> df = ks.DataFrame({
...     'col1': ['A', 'A', 'B', None, 'D', 'C'],
...     'col2': [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
... },
...     columns=['col1', 'col2', 'col3'])
>>> df.sort_values(by=['col1', 'col2'])
   col1  col2  col3
1     A     1     1
0     A     2     0
2     B     9     9
5     C     4     3
4     D     7     2
3  None     8     4
```

**databricks.koalas.DataFrame.nlargest**

`DataFrame.nlargest` (*n*: int, *columns*: Any) → databricks.koalas.frame.DataFrame

Return the first *n* rows ordered by *columns* in descending order.

Return the first *n* rows with the largest values in *columns*, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=False).head(n)`, but more performant in pandas. In Koalas, thanks to Spark's lazy execution and query optimizer, the two would have same performance.

**Parameters**

**n** [int] Number of rows to return.

**columns** [label or list of labels] Column label(s) to order by.

**Returns**

**DataFrame** The first *n* rows ordered by the given columns in descending order.

See also:

**`DataFrame.nsmallest`** Return the first *n* rows ordered by *columns* in ascending order.

**`DataFrame.sort_values`** Sort DataFrame by the values.

**`DataFrame.head`** Return the first  $n$  rows without re-ordering.

## Notes

This function cannot be used with all column types. For example, when specifying columns with *object* or *category* dtypes, `TypeError` is raised.

## Examples

```
>>> df = ks.DataFrame({'X': [1, 2, 3, 5, 6, 7, np.nan],
...                    'Y': [6, 7, 8, 9, 10, 11, 12]})
>>> df
   X  Y
0  1.0  6
1  2.0  7
2  3.0  8
3  5.0  9
4  6.0 10
5  7.0 11
6  NaN 12
```

In the following example, we will use `nlargest` to select the three rows having the largest values in column “population”.

```
>>> df.nlargest(n=3, columns='X')
   X  Y
5  7.0 11
4  6.0 10
3  5.0  9
```

```
>>> df.nlargest(n=3, columns=['Y', 'X'])
   X  Y
6  NaN 12
5  7.0 11
4  6.0 10
```

## `databricks.koalas.DataFrame.nsmallest`

`DataFrame.nsmallest` ( $n$ : *int*, *columns*: *Any*) → `databricks.koalas.frame.DataFrame`

Return the first  $n$  rows ordered by *columns* in ascending order.

Return the first  $n$  rows with the smallest values in *columns*, in ascending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=True).head(n)`, but more performant. In Koalas, thanks to Spark’s lazy execution and query optimizer, the two would have same performance.

### Parameters

**`n`** [*int*] Number of items to retrieve.

**`columns`** [*list* or *str*] Column name or names to order by.

**Returns****DataFrame**

See also:

**DataFrame.nlargest** Return the first *n* rows ordered by *columns* in descending order.

**DataFrame.sort\_values** Sort DataFrame by the values.

**DataFrame.head** Return the first *n* rows without re-ordering.

**Examples**

```
>>> df = ks.DataFrame({'X': [1, 2, 3, 5, 6, 7, np.nan],
...                    'Y': [6, 7, 8, 9, 10, 11, 12]})
>>> df
   X    Y
0  1.0  6
1  2.0  7
2  3.0  8
3  5.0  9
4  6.0 10
5  7.0 11
6  NaN 12
```

In the following example, we will use `nsmallest` to select the three rows having the smallest values in column “a”.

```
>>> df.nsmallest(n=3, columns='X')
   X    Y
0  1.0  6
1  2.0  7
2  3.0  8
```

To order by the largest values in column “a” and then “c”, we can specify multiple columns like in the next example.

```
>>> df.nsmallest(n=3, columns=['Y', 'X'])
   X    Y
0  1.0  6
1  2.0  7
2  3.0  8
```

**databricks.koalas.DataFrame.stack**

`DataFrame.stack()` → Union[DataFrame, Series]

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

- if the columns have a single level, the output is a Series;
- if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

The new index levels are sorted.

### Returns

**DataFrame or Series** Stacked dataframe or series.

See also:

**DataFrame.unstack** Unstack prescribed level(s) from index axis onto column axis.

**DataFrame.pivot** Reshape dataframe from long format to wide format.

**DataFrame.pivot\_table** Create a spreadsheet-style pivot table as a DataFrame.

### Notes

The function is named by analogy with a collection of books being reorganized from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of each other (in the index of the dataframe).

### Examples

#### Single level columns

```
>>> df_single_level_cols = ks.DataFrame([[0, 1], [2, 3]],
...                                     index=['cat', 'dog'],
...                                     columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
   weight  height
cat      0      1
dog      2      3
>>> df_single_level_cols.stack().sort_index()
cat  height      1
     weight      0
dog  height      3
     weight      2
dtype: int64
```

#### Multi level columns: simple case

```
>>> multicoll = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                       ('weight', 'pounds')])
>>> df_multi_level_cols1 = ks.DataFrame([[1, 2], [2, 4]],
...                                     index=['cat', 'dog'],
...                                     columns=multicoll)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
   weight
      kg  pounds
cat    1      2
dog    2      4
>>> df_multi_level_cols1.stack().sort_index()
```

(continues on next page)

(continued from previous page)

		weight
cat	kg	1
	pounds	2
dog	kg	2
	pounds	4

**Missing values**

```
>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                     ('height', 'm')])
>>> df_multi_level_cols2 = ks.DataFrame([[1.0, 2.0], [3.0, 4.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
...                                     columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
   weight height
   kg      m
cat   1.0    2.0
dog   3.0    4.0
>>> df_multi_level_cols2.stack().sort_index()
   height weight
cat kg    NaN   1.0
   m     2.0   NaN
dog kg    NaN   3.0
   m     4.0   NaN
```

**databricks.koalas.DataFrame.unstack**

`DataFrame.unstack()` → Union[DataFrame, Series]

Pivot the (necessarily hierarchical) index labels.

Returns a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.

If the index is not a MultiIndex, the output will be a Series.

---

**Note:** If the index is a MultiIndex, the output DataFrame could be very wide, and it could cause a serious performance degradation since Spark partitions it row based.

---

**Returns****Series or DataFrame**

See also:

**`DataFrame.pivot`** Pivot a table based on column values.

**`DataFrame.stack`** Pivot a level of the column labels (inverse operation from unstack).

## Examples

```
>>> df = ks.DataFrame({"A": {"0": "a", "1": "b", "2": "c"},
...                    "B": {"0": "1", "1": "3", "2": "5"},
...                    "C": {"0": "2", "1": "4", "2": "6"}},
...                   columns=["A", "B", "C"])
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.unstack().sort_index()
A  0    a
   1    b
   2    c
B  0    1
   1    3
   2    5
C  0    2
   1    4
   2    6
dtype: object
```

```
>>> df.columns = pd.MultiIndex.from_tuples([('X', 'A'), ('X', 'B'), ('Y', 'C')])
>>> df.unstack().sort_index()
X  A  0    a
   1    b
   2    c
   B  0    1
   1    3
   2    5
Y  C  0    2
   1    4
   2    6
dtype: object
```

For MultiIndex case:

```
>>> df = ks.DataFrame({"A": ["a", "b", "c"],
...                    "B": [1, 3, 5],
...                    "C": [2, 4, 6]},
...                   columns=["A", "B", "C"])
>>> df = df.set_index('A', append=True)
>>> df
   B  C
A
0 a  1  2
1 b  3  4
2 c  5  6
>>> df.unstack().sort_index()
   B      C
A  a  b  c  a  b  c
0  1.0 NaN NaN  2.0 NaN NaN
1  NaN  3.0 NaN NaN  4.0 NaN
2  NaN NaN  5.0 NaN NaN  6.0
```



**databricks.koalas.DataFrame.melt**

`DataFrame.melt` (*id\_vars=None*, *value\_vars=None*, *var\_name=None*, *value\_name='value'*) → `databricks.koalas.frame.DataFrame`

Unpivot a DataFrame from wide format to long format, optionally leaving identifier variables set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id\_vars*), while all other columns, considered measured variables (*value\_vars*), are “unpivoted” to the row axis, leaving just two non-identifier columns, ‘variable’ and ‘value’.

**Parameters**

**frame** [DataFrame]

**id\_vars** [tuple, list, or ndarray, optional] Column(s) to use as identifier variables.

**value\_vars** [tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as *id\_vars*.

**var\_name** [scalar, default ‘variable’] Name to use for the ‘variable’ column. If None it uses *frame.columns.name* or ‘variable’.

**value\_name** [scalar, default ‘value’] Name to use for the ‘value’ column.

**Returns**

**DataFrame** Unpivoted DataFrame.

**Examples**

```
>>> df = ks.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                    'B': {0: 1, 1: 3, 2: 5},
...                    'C': {0: 2, 1: 4, 2: 6}},
...                    columns=['A', 'B', 'C'])
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> ks.melt(df)
   variable  value
0         A      a
1         B      1
2         C      2
3         A      b
4         B      3
5         C      4
6         A      c
7         B      5
8         C      6
```

```
>>> df.melt(id_vars='A')
   A variable  value
0  a         B      1
1  a         C      2
2  b         B      3
3  b         C      4
```

(continues on next page)

(continued from previous page)

4	c	B	5
5	c	C	6

```
>>> df.melt(value_vars='A')
  variable value
0        A     a
1        A     b
2        A     c
```

```
>>> ks.melt(df, id_vars=['A', 'B'])
   A  B variable  value
0  a  1         C      2
1  b  3         C      4
2  c  5         C      6
```

```
>>> df.melt(id_vars=['A'], value_vars=['C'])
   A variable  value
0  a         C      2
1  b         C      4
2  c         C      6
```

The names of ‘variable’ and ‘value’ columns can be customized:

```
>>> ks.melt(df, id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname myValname
0  a          B          1
1  b          B          3
2  c          B          5
```

## databricks.koalas.DataFrame.explode

`DataFrame.explode(column) → databricks.koalas.frame.DataFrame`  
 Transform each element of a list-like to a row, replicating index values.

### Parameters

**column** [str or tuple] Column to explode.

### Returns

**DataFrame** Exploded lists to rows of the subset columns; index will be duplicated for these rows.

### See also:

**DataFrame.unstack** Pivot a level of the (necessarily hierarchical) index labels.

**DataFrame.melt** Unpivot a DataFrame from wide format to long format.

## Examples

```
>>> df = ks.DataFrame({'A': [[1, 2, 3], [], [3, 4]], 'B': 1})
>>> df
```

	A	B
0	[1, 2, 3]	1
1	[]	1
2	[3, 4]	1

```
>>> df.explode('A')
```

	A	B
0	1.0	1
0	2.0	1
0	3.0	1
1	NaN	1
2	3.0	1
2	4.0	1

## databricks.koalas.DataFrame.squeeze

`DataFrame.squeeze(axis=None)` → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, DataFrame, Series]

Squeeze 1 dimensional axis objects into scalars.

Series or DataFrames with a single element are squeezed to a scalar. DataFrames with a single column or a single row are squeezed to a Series. Otherwise the object is unchanged.

This method is most useful when you don't know if your object is a Series or DataFrame, but you do know it has just a single column. In that case you can safely call *squeeze* to ensure you have a Series.

### Parameters

**axis** [{0 or 'index', 1 or 'columns', None}, default None] A specific axis to squeeze. By default, all length-1 axes are squeezed.

### Returns

**DataFrame, Series, or scalar** The projection after squeezing *axis* or all the axes.

See also:

**Series.iloc** Integer-location based indexing for selecting scalars.

**DataFrame.iloc** Integer-location based indexing for selecting Series.

**Series.to\_frame** Inverse of DataFrame.squeeze for a single-column DataFrame.

## Examples

```
>>> primes = ks.Series([2, 3, 5, 7])
```

Slicing might produce a Series with a single value:

```
>>> even_primes = primes[primes % 2 == 0]
>>> even_primes
```

	even_primes
0	2

dtype: int64

```
>>> even_primes.squeeze()
2
```

Squeezing objects with more than one value in every axis does nothing:

```
>>> odd_primes = primes[primes % 2 == 1]
>>> odd_primes
1    3
2    5
3    7
dtype: int64
```

```
>>> odd_primes.squeeze()
1    3
2    5
3    7
dtype: int64
```

Squeezing is even more effective when used with DataFrames.

```
>>> df = ks.DataFrame([[1, 2], [3, 4]], columns=['a', 'b'])
>>> df
   a  b
0  1  2
1  3  4
```

Slicing a single column will produce a DataFrame with the columns having only one value:

```
>>> df_a = df[['a']]
>>> df_a
   a
0  1
1  3
```

So the columns can be squeezed down, resulting in a Series:

```
>>> df_a.squeeze('columns')
0    1
1    3
Name: a, dtype: int64
```

Slicing a single row from a single column will produce a single scalar DataFrame:

```
>>> df_1a = df.loc[[1], ['a']]
>>> df_1a
   a
1  3
```

Squeezing the rows produces a single scalar Series:

```
>>> df_1a.squeeze('rows')
a    3
Name: 1, dtype: int64
```

Squeezing all axes will project directly into a scalar:

```
>>> df_1a.squeeze()
3
```

## databricks.koalas.DataFrame.T

**property** DataFrame.T

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method `transpose()`.

**Note:** This method is based on an expensive operation due to the nature of big data. Internally it needs to generate each row for each value, and then group twice - it is a huge operation. To prevent misuse, this method has the 'compute.max\_rows' default limit of input length, and raises a ValueError.

```
>>> from databricks.koalas.config import option_context
>>> with option_context('compute.max_rows', 1000):
...     ks.DataFrame({'a': range(1001)}).transpose()
Traceback (most recent call last):
...
ValueError: Current DataFrame has more then the given limit 1000 rows.
Please set 'compute.max_rows' by using 'databricks.koalas.config.set_option'
to retrieve to retrieve more than 1000 rows. Note that, before changing the
'compute.max_rows', this operation is considerably expensive.
```

## Returns

**DataFrame** The transposed DataFrame.

## Notes

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the coerced dtype. For instance, if int and float have to be placed in same column, it becomes float. If type coercion is not possible, it fails.

Also, note that the values in index should be unique because they become unique column names.

In addition, if Spark 2.3 is used, the types should always be exactly same.

## Examples

### Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = ks.DataFrame(data=d1, columns=['col1', 'col2'])
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T.sort_index()
>>> df1_transposed
      0  1
col1  1  2
col2  3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

### Non-square DataFrame with mixed dtypes

```
>>> d2 = {'score': [9.5, 8],
...      'kids': [0, 0],
...      'age': [12, 22]}
>>> df2 = ks.DataFrame(data=d2, columns=['score', 'kids', 'age'])
>>> df2
   score  kids  age
0    9.5     0   12
1    8.0     0   22
```

```
>>> df2_transposed = df2.T.sort_index()
>>> df2_transposed
      0  1
age   12.0 22.0
kids   0.0  0.0
score   9.5  8.0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the coerced dtype:

```
>>> df2.dtypes
score    float64
kids      int64
age      int64
dtype: object
```

```
>>> df2_transposed.dtypes
0    float64
1    float64
dtype: object
```

**databricks.koalas.DataFrame.transpose**

`DataFrame.transpose()` → `databricks.koalas.frame.DataFrame`

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property `T` is an accessor to the method `transpose()`.

**Note:** This method is based on an expensive operation due to the nature of big data. Internally it needs to generate each row for each value, and then group twice - it is a huge operation. To prevent misuse, this method has the 'compute.max\_rows' default limit of input length, and raises a `ValueError`.

```
>>> from databricks.koalas.config import option_context
>>> with option_context('compute.max_rows', 1000):
...     ks.DataFrame({'a': range(1001)}).transpose()
Traceback (most recent call last):
...
ValueError: Current DataFrame has more then the given limit 1000 rows.
Please set 'compute.max_rows' by using 'databricks.koalas.config.set_option'
to retrieve to retrieve more than 1000 rows. Note that, before changing the
'compute.max_rows', this operation is considerably expensive.
```

**Returns**

**DataFrame** The transposed DataFrame.

**Notes**

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the coerced dtype. For instance, if int and float have to be placed in same column, it becomes float. If type coercion is not possible, it fails.

Also, note that the values in index should be unique because they become unique column names.

In addition, if Spark 2.3 is used, the types should always be exactly same.

**Examples****Square DataFrame with homogeneous dtype**

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = ks.DataFrame(data=d1, columns=['col1', 'col2'])
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T.sort_index()
>>> df1_transposed
   0  1
col1 1  2
col2 3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

### Non-square DataFrame with mixed dtypes

```
>>> d2 = {'score': [9.5, 8],
...      'kids': [0, 0],
...      'age': [12, 22]}
>>> df2 = ks.DataFrame(data=d2, columns=['score', 'kids', 'age'])
>>> df2
   score  kids  age
0   9.5    0   12
1   8.0    0   22
```

```
>>> df2_transposed = df2.T.sort_index()
>>> df2_transposed
      0    1
age  12.0  22.0
kids   0.0   0.0
score   9.5   8.0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the coerced dtype:

```
>>> df2.dtypes
score    float64
kids      int64
age      int64
dtype: object
```

```
>>> df2_transposed.dtypes
0    float64
1    float64
dtype: object
```

### `databricks.koalas.DataFrame.reindex`

`DataFrame.reindex(labels: Optional[Any] = None, index: Optional[Any] = None, columns: Optional[Any] = None, axis: Union[str, int, None] = None, copy: Optional[bool] = True, fill_value: Optional[Any] = None) → databricks.koalas.frame.DataFrame`

Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`.

#### Parameters

**labels: array-like, optional** New labels / index to conform the axis specified by ‘axis’ to.

**index, columns: array-like, optional** New labels / index to conform to, should be specified using keywords. Preferably an Index object to avoid duplicating data



**axis:** `int or str, optional` Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).

**copy** [`bool`, default `True`] Return a new object, even if the passed indexes are the same.

**fill\_value** [`scalar`, default `np.NaN`] Value to use for missing values. Defaults to `NaN`, but can be any “compatible” value.

### Returns

**DataFrame with changed index.**

See also:

`DataFrame.set_index` Set row labels.

`DataFrame.reset_index` Remove row labels or move them to new columns.

### Examples

`DataFrame.reindex` supports two calling conventions

- `(index=index_labels, columns=column_labels, ...)`
- `(labels, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = ks.DataFrame({
...     'http_status': [200, 200, 404, 404, 301],
...     'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...     index=index,
...     columns=['http_status', 'response_time'])
>>> df
```

	http_status	response_time
Firefox	200	0.04
Chrome	200	0.02
Safari	404	0.07
IE10	404	0.08
Konqueror	301	1.00

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned `NaN`.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index).sort_index()
```

	http_status	response_time
Chrome	200.0	0.02
Comodo Dragon	NaN	NaN
IE10	404.0	0.08
Iceweasel	NaN	NaN
Safari	404.0	0.07

We can fill in the missing values by passing a value to the keyword `fill_value`.

```
>>> df.reindex(new_index, fill_value=0, copy=False).sort_index()
           http_status  response_time
Chrome                200            0.02
Comodo Dragon          0            0.00
IE10                  404            0.08
Iceweasel              0            0.00
Safari                404            0.07
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent']).sort_index()
           http_status  user_agent
Chrome                200         NaN
Firefox               200         NaN
IE10                  404         NaN
Konqueror             301         NaN
Safari                404         NaN
```

Or we can use “axis-style” keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns").sort_index()
           http_status  user_agent
Chrome                200         NaN
Firefox               200         NaN
IE10                  404         NaN
Konqueror             301         NaN
Safari                404         NaN
```

To further illustrate the filling functionality in reindex, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = ks.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                     index=date_index)
>>> df2.sort_index()
           prices
2010-01-01    100.0
2010-01-02    101.0
2010-01-03     NaN
2010-01-04    100.0
2010-01-05     89.0
2010-01-06     88.0
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2).sort_index()
           prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01    100.0
2010-01-02    101.0
2010-01-03     NaN
2010-01-04    100.0
2010-01-05     89.0
2010-01-06     88.0
2010-01-07     NaN
```

**databricks.koalas.DataFrame.reindex\_like**

`DataFrame.reindex_like` (*other*: `databricks.koalas.frame.DataFrame`, *copy*: `bool = True`) → `databricks.koalas.frame.DataFrame`

Return a DataFrame with matching indices as other object.

Conform the object to the same index on all axes. Places NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`.

**Parameters**

**other** [DataFrame] Its row and column indices are used to define the new indices of this object.

**copy** [bool, default True] Return a new object, even if the passed indexes are the same.

**Returns**

**DataFrame** DataFrame with changed indices on each axis.

See also:

`DataFrame.set_index` Set row labels.

`DataFrame.reset_index` Remove row labels or move them to new columns.

`DataFrame.reindex` Change to new indices or expand indices.

**Notes**

Same as calling `.reindex(index=other.index, columns=other.columns, ...)`.

**Examples**

```
>>> df1 = ks.DataFrame([[24.3, 75.7, 'high'],
...                     [31, 87.8, 'high'],
...                     [22, 71.6, 'medium'],
...                     [35, 95, 'medium']],
...                     columns=['temp_celsius', 'temp_fahrenheit',
...                               'windspeed'],
...                     index=pd.date_range(start='2014-02-12',
...                                           end='2014-02-15', freq='D'))
>>> df1
```

	temp_celsius	temp_fahrenheit	windspeed
2014-02-12	24.3	75.7	high
2014-02-13	31.0	87.8	high
2014-02-14	22.0	71.6	medium
2014-02-15	35.0	95.0	medium

```
>>> df2 = ks.DataFrame([[28, 'low'],
...                     [30, 'low'],
...                     [35.1, 'medium']],
...                     columns=['temp_celsius', 'windspeed'],
...                     index=pd.DatetimeIndex(['2014-02-12', '2014-02-13',
...                                               '2014-02-15']))
>>> df2
```

	temp_celsius	windspeed
2014-02-12	28.0	low

(continues on next page)

(continued from previous page)

2014-02-13	30.0	low
2014-02-15	35.1	medium

```
>>> df2.reindex_like(df1).sort_index()
      temp_celsius  temp_fahrenheit  windspeed
2014-02-12      28.0             NaN         low
2014-02-13      30.0             NaN         low
2014-02-14       NaN             NaN         None
2014-02-15      35.1             NaN        medium
```

## databricks.koalas.DataFrame.rank

`DataFrame.rank(method='average', ascending=True)` → `databricks.koalas.frame.DataFrame`

Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values.

**Note:** the current implementation of rank uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

### Parameters

**method** [{ 'average', 'min', 'max', 'first', 'dense' }]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

**ascending** [boolean, default True] False for ranks by high (1) to low (N)

### Returns

**ranks** [same type as caller]

## Examples

```
>>> df = ks.DataFrame({'A': [1, 2, 2, 3], 'B': [4, 3, 2, 1]}, columns= ['A', 'B'])
>>> df
   A  B
0  1  4
1  2  3
2  2  2
3  3  1
```

```
>>> df.rank().sort_index()
      A    B
0  1.0  4.0
1  2.5  3.0
```

(continues on next page)

(continued from previous page)

```
2  2.5  2.0
3  4.0  1.0
```

If method is set to 'min', it use lowest rank in group.

```
>>> df.rank(method='min').sort_index()
      A      B
0  1.0  4.0
1  2.0  3.0
2  2.0  2.0
3  4.0  1.0
```

If method is set to 'max', it use highest rank in group.

```
>>> df.rank(method='max').sort_index()
      A      B
0  1.0  4.0
1  3.0  3.0
2  3.0  2.0
3  4.0  1.0
```

If method is set to 'dense', it leaves no gaps in group.

```
>>> df.rank(method='dense').sort_index()
      A      B
0  1.0  4.0
1  2.0  3.0
2  2.0  2.0
3  3.0  1.0
```

### 3.4.11 Combining / joining / merging

<code>DataFrame.append(other[, ignore_index, ...])</code>	Append rows of other to the end of caller, returning a new object.
<code>DataFrame.assign(**kwargs)</code>	Assign new columns to a DataFrame.
<code>DataFrame.merge(right[, how, on, left_on, ...])</code>	Merge DataFrame objects with a database-style join.
<code>DataFrame.join(right[, on, how, lsuffix, ...])</code>	Join columns of another DataFrame.
<code>DataFrame.update(other[, join, overwrite])</code>	Modify in place using non-NA values from another DataFrame.
<code>DataFrame.insert(loc, column, value[, ...])</code>	Insert column into DataFrame at specified location.

#### **databricks.koalas.DataFrame.append**

`DataFrame.append(other: databricks.koalas.frame.DataFrame, ignore_index: bool = False, verify_integrity: bool = False, sort: bool = False) → databricks.koalas.frame.DataFrame`  
Append rows of other to the end of caller, returning a new object.

Columns in other that are not in the caller are added as new columns.

##### **Parameters**

**other** [DataFrame or Series/dict-like object, or list of these] The data to append.

**ignore\_index** [boolean, default False] If True, do not use the index labels.

**verify\_integrity** [boolean, default False] If True, raise ValueError on creating index with duplicates.

**sort** [boolean, default False] Currently not supported.

### Returns

**appended** [DataFrame]

### Examples

```
>>> df = ks.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
```

```
>>> df.append(df)
  A  B
0  1  2
1  3  4
0  1  2
1  3  4
```

```
>>> df.append(df, ignore_index=True)
  A  B
0  1  2
1  3  4
2  1  2
3  3  4
```

### `databricks.koalas.DataFrame.assign`

`DataFrame.assign(**kwargs)` → `databricks.koalas.frame.DataFrame`

Assign new columns to a DataFrame.

Returns a new object with all original columns in addition to new ones. Existing columns that are re-assigned will be overwritten.

### Parameters

**\*\*kwargs** [dict of {str: callable, Series or Index}] The column names are keywords. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though Koalas doesn't check it). If the values are not callable, (e.g. a Series or a literal), they are simply assigned.

### Returns

**DataFrame** A new DataFrame with the new columns in addition to all the existing columns.

## Notes

Assigning multiple columns within the same `assign` is possible but you cannot refer to newly created or modified columns. This feature is supported in pandas for Python 3.6 and later but not in Koalas. In Koalas, all items are computed first, and then assigned.

## Examples

```
>>> df = ks.DataFrame({'temp_c': [17.0, 25.0]},
...                    index=['Portland', 'Berkeley'])
>>> df
```

	temp_c
Portland	17.0
Berkeley	25.0

Where the value is a callable, evaluated on `df`:

```
>>> df.assign(temp_f=lambda x: x.temp_c * 9 / 5 + 32)
```

	temp_c	temp_f
Portland	17.0	62.6
Berkeley	25.0	77.0

Alternatively, the same behavior can be achieved by directly referencing an existing Series or sequence and you can also create multiple columns within the same assign.

```
>>> assigned = df.assign(temp_f=df['temp_c'] * 9 / 5 + 32,
...                      temp_k=df['temp_c'] + 273.15,
...                      temp_idx=df.index)
>>> assigned[['temp_c', 'temp_f', 'temp_k', 'temp_idx']]
```

	temp_c	temp_f	temp_k	temp_idx
Portland	17.0	62.6	290.15	Portland
Berkeley	25.0	77.0	298.15	Berkeley

## databricks.koalas.DataFrame.merge

`DataFrame.merge(right: databricks.koalas.frame.DataFrame, how: str = 'inner', on: Union[Any, List[Any], Tuple, List[Tuple], None] = None, left_on: Union[Any, List[Any], Tuple, List[Tuple], None] = None, right_on: Union[Any, List[Any], Tuple, List[Tuple], None] = None, left_index: bool = False, right_index: bool = False, suffixes: Tuple[str, str] = ('_x', '_y')) → databricks.koalas.frame.DataFrame`

Merge DataFrame objects with a database-style join.

**The index of the resulting DataFrame will be one of the following:**

- `0...n` if no index is used for merging
- Index of the left DataFrame if merged only on the index of the right DataFrame
- Index of the right DataFrame if merged only on the index of the left DataFrame
- **All involved indices if merged using the indices of both DataFrames** e.g. if *left* with indices (a, x) and *right* with indices (b, x), the result will be an index (x, a, b)

### Parameters

**right:** Object to merge with.

**how:** Type of merge to be performed. {'left', 'right', 'outer', 'inner'}, default 'inner'

**left:** use only keys from left frame, similar to a SQL left outer join; not preserve key order unlike pandas.

**right:** use only keys from right frame, similar to a SQL right outer join; not preserve key order unlike pandas.

**outer:** use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.

**inner:** use intersection of keys from both frames, similar to a SQL inner join; not preserve the order of the left keys unlike pandas.

**on:** Column or index level names to join on. These must be found in both DataFrames. If on is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

**left\_on:** Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

**right\_on:** Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

**left\_index:** Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

**right\_index:** Use the index from the right DataFrame as the join key. Same caveats as left\_index.

**suffixes:** Suffix to apply to overlapping column names in the left and right side, respectively.

### Returns

**DataFrame** A DataFrame of the two merged objects.

### See also:

*DataFrame.join* Join columns of another DataFrame.

*DataFrame.update* Modify in place using non-NA values from another DataFrame.

*DataFrame.hint* Specifies some hint on the current DataFrame.

*broadcast* Marks a DataFrame as small enough for use in broadcast joins.

### Notes

As described in #263, joining string columns currently returns None for missing values instead of NaN.



## Examples

```
>>> df1 = ks.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [1, 2, 3, 5]},
...                     columns=['lkey', 'value'])
>>> df2 = ks.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [5, 6, 7, 8]},
...                     columns=['rkey', 'value'])
>>> df1
   lkey  value
0  foo      1
1  bar      2
2  baz      3
3  foo      5
>>> df2
   rkey  value
0  foo      5
1  bar      6
2  baz      7
3  foo      8
```

Merge df1 and df2 on the lkey and rkey columns. The value columns have the default suffixes, \_x and \_y, appended.

```
>>> merged = df1.merge(df2, left_on='lkey', right_on='rkey')
>>> merged.sort_values(by=['lkey', 'value_x', 'rkey', 'value_y'])
   lkey  value_x rkey  value_y
...bar      2  bar      6
...baz      3  baz      7
...foo      1  foo      5
...foo      1  foo      8
...foo      5  foo      5
...foo      5  foo      8
```

```
>>> left_kdf = ks.DataFrame({'A': [1, 2]})
>>> right_kdf = ks.DataFrame({'B': ['x', 'y']}, index=[1, 2])
```

```
>>> left_kdf.merge(right_kdf, left_index=True, right_index=True).sort_index()
   A  B
1  2  x
```

```
>>> left_kdf.merge(right_kdf, left_index=True, right_index=True, how='left').sort_index()
   A  B
0  1  None
1  2   x
```

```
>>> left_kdf.merge(right_kdf, left_index=True, right_index=True, how='right').sort_index()
   A  B
1  2.0  x
2  NaN  y
```

```
>>> left_kdf.merge(right_kdf, left_index=True, right_index=True, how='outer').sort_index()
   A  B
```

(continues on next page)

(continued from previous page)

0	1.0	None
1	2.0	x
2	NaN	y

**databricks.koalas.DataFrame.join**

`DataFrame.join(right: databricks.koalas.frame.DataFrame, on: Union[Any, List[Any], Tuple, List[Tuple], None] = None, how: str = 'left', lsuffix: str = "", rsuffix: str = "") → databricks.koalas.frame.DataFrame`  
Join columns of another DataFrame.

Join columns with *right* DataFrame either on index or on a key column. Efficiently join multiple DataFrame objects by index at once by passing a list.

**Parameters**

**right:** DataFrame, Series

**on:** str, list of str, or array-like, optional Column or index level name(s) in the caller to join on the index in *right*, otherwise joins index-on-index. If multiple values given, the *right* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation.

**how:** {'left', 'right', 'outer', 'inner'}, default 'left' How to handle the operation of the two objects.

- left: use *left* frame's index (or column if on is specified).
- right: use *right*'s index.
- outer: form union of *left* frame's index (or column if on is specified) with *right*'s index, and sort it. lexicographically.
- inner: form intersection of *left* frame's index (or column if on is specified) with *right*'s index, preserving the order of the *left*'s one.

**lsuffix** [str, default ''] Suffix to use from left frame's overlapping columns.

**rsuffix** [str, default ''] Suffix to use from *right* frame's overlapping columns.

**Returns**

**DataFrame** A dataframe containing columns from both the *left* and *right*.

**See also:**

**DataFrame.merge** For column(s)-on-columns(s) operations.

**DataFrame.update** Modify in place using non-NA values from another DataFrame.

**DataFrame.hint** Specifies some hint on the current DataFrame.

**broadcast** Marks a DataFrame as small enough for use in broadcast joins.

## Notes

Parameters `on`, `lsuffix`, and `rsuffix` are not supported when passing a list of DataFrame objects.

## Examples

```
>>> kdf1 = ks.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
...                      'A': ['A0', 'A1', 'A2', 'A3']},
...                      columns=['key', 'A'])
>>> kdf2 = ks.DataFrame({'key': ['K0', 'K1', 'K2'],
...                      'B': ['B0', 'B1', 'B2']},
...                      columns=['key', 'B'])
>>> kdf1
   key  A
0  K0  A0
1  K1  A1
2  K2  A2
3  K3  A3
>>> kdf2
   key  B
0  K0  B0
1  K1  B1
2  K2  B2
```

Join DataFrames using their indexes.

```
>>> join_kdf = kdf1.join(kdf2, lsuffix='_left', rsuffix='_right')
>>> join_kdf.sort_values(by=join_kdf.columns)
   key_left  A key_right  B
0      K0  A0      K0  B0
1      K1  A1      K1  B1
2      K2  A2      K2  B2
3      K3  A3      None None
```

If we want to join using the key columns, we need to set `key` to be the index in both `df` and `right`. The joined DataFrame will have `key` as its index.

```
>>> join_kdf = kdf1.set_index('key').join(kdf2.set_index('key'))
>>> join_kdf.sort_values(by=join_kdf.columns)
   A      B
key
K0  A0    B0
K1  A1    B1
K2  A2    B2
K3  A3    None
```

Another option to join using the key columns is to use the `on` parameter. `DataFrame.join` always uses `right`'s index but we can use any column in `df`. This method not preserve the original DataFrame's index in the result unlike `pandas`.

```
>>> join_kdf = kdf1.join(kdf2.set_index('key'), on='key')
>>> join_kdf.index
Int64Index([0, 1, 2, 3], dtype='int64')
```

**databricks.koalas.DataFrame.update**

`DataFrame.update` (*other*: `databricks.koalas.frame.DataFrame`, *join*: `str = 'left'`, *overwrite*: `bool = True`)  
     → `None`

Modify in place using non-NA values from another DataFrame. Aligns on indices. There is no return value.

**Parameters**

**other** [DataFrame, or Series]

**join** ['left', default 'left'] Only left join is implemented, keeping the index and columns of the original object.

**overwrite** [bool, default True] How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.
- False: only update values that are NA in the original DataFrame.

**Returns**

`None` [method directly changes calling object]

See also:

`DataFrame.merge` For column(s)-on-columns(s) operations.

`DataFrame.join` Join columns of another DataFrame.

`DataFrame.hint` Specifies some hint on the current DataFrame.

`broadcast` Marks a DataFrame as small enough for use in broadcast joins.

**Examples**

```
>>> df = ks.DataFrame({'A': [1, 2, 3], 'B': [400, 500, 600]}, columns=['A', 'B'])
>>> new_df = ks.DataFrame({'B': [4, 5, 6], 'C': [7, 8, 9]}, columns=['B', 'C'])
>>> df.update(new_df)
>>> df.sort_index()
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = ks.DataFrame({'A': ['a', 'b', 'c'], 'B': ['x', 'y', 'z']}, columns=['A',
↳ 'B'])
>>> new_df = ks.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']}, columns=['B'])
>>> df.update(new_df)
>>> df.sort_index()
   A  B
0  a  d
1  b  e
2  c  f
```

For Series, it's name attribute must be set.

```
>>> df = ks.DataFrame({'A': ['a', 'b', 'c'], 'B': ['x', 'y', 'z']}, columns=['A',
↳ 'B'])
>>> new_column = ks.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df.sort_index()
   A  B
0  a  d
1  b  y
2  c  e
```

If *other* contains None the corresponding values are not updated in the original dataframe.

```
>>> df = ks.DataFrame({'A': [1, 2, 3], 'B': [400, 500, 600]}, columns=['A', 'B'])
>>> new_df = ks.DataFrame({'B': [4, None, 6]}, columns=['B'])
>>> df.update(new_df)
>>> df.sort_index()
   A      B
0  1    4.0
1  2  500.0
2  3    6.0
```

### databricks.koalas.DataFrame.insert

`DataFrame.insert` (*loc*: int, *column*, *value*: Union[int, float, decimal.Decimal, datetime.date, None, Series, Iterable], *allow\_duplicates*: bool = False) → None

Insert column into DataFrame at specified location.

Raises a ValueError if *column* is already contained in the DataFrame, unless *allow\_duplicates* is set to True.

#### Parameters

**loc** [int] Insertion index. Must verify  $0 \leq \text{loc} \leq \text{len}(\text{columns})$ .

**column** [str, number, or hashable object] Label of the inserted column.

**value** [int, Series, or array-like]

**allow\_duplicates** [bool, optional]

#### Examples

```
>>> kdf = ks.DataFrame([1, 2, 3])
>>> kdf.sort_index()
   0
0  1
1  2
2  3
>>> kdf.insert(0, 'x', 4)
>>> kdf.sort_index()
   x  0
0  4  1
1  4  2
2  4  3
```

```
>>> from databricks.koalas.config import set_option, reset_option
>>> set_option("compute.ops_on_diff_frames", True)
```

```
>>> kdf.insert(1, 'y', [5, 6, 7])
>>> kdf.sort_index()
   x  y  0
0  4  5  1
1  4  6  2
2  4  7  3
```

```
>>> kdf.insert(2, 'z', ks.Series([8, 9, 10]))
>>> kdf.sort_index()
   x  y  z  0
0  4  5  8  1
1  4  6  9  2
2  4  7 10  3
```

```
>>> reset_option("compute.ops_on_diff_frames")
```

### 3.4.12 Time series-related

<code>DataFrame.shift([periods, fill_value])</code>	Shift DataFrame by desired number of periods.
<code>DataFrame.first_valid_index()</code>	Retrieves the index of the first valid value.
<code>DataFrame.last_valid_index()</code>	Return index for last non-NA/null value.

#### **databricks.koalas.DataFrame.shift**

`DataFrame.shift` (*periods=1, fill\_value=None*) → `databricks.koalas.frame.DataFrame`  
Shift DataFrame by desired number of periods.

**Note:** the current implementation of shift uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

#### **Parameters**

**periods** [int] Number of periods to shift. Can be positive or negative.

**fill\_value** [object, optional] The scalar value to use for newly introduced missing values. The default depends on the dtype of self. For numeric data, `np.nan` is used.

#### **Returns**

Copy of input DataFrame, shifted.

## Examples

```
>>> df = ks.DataFrame({'Col1': [10, 20, 15, 30, 45],
...                     'Col2': [13, 23, 18, 33, 48],
...                     'Col3': [17, 27, 22, 37, 52]},
...                     columns=['Col1', 'Col2', 'Col3'])
```

```
>>> df.shift(periods=3)
   Col1  Col2  Col3
0   NaN   NaN   NaN
1   NaN   NaN   NaN
2   NaN   NaN   NaN
3  10.0  13.0  17.0
4  20.0  23.0  27.0
```

```
>>> df.shift(periods=3, fill_value=0)
   Col1  Col2  Col3
0     0     0     0
1     0     0     0
2     0     0     0
3    10    13    17
4    20    23    27
```

## databricks.koalas.DataFrame.first\_valid\_index

`DataFrame.first_valid_index()` → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Tuple[Union[int, float, str, bytes, decimal.Decimal, datetime.date, None], ...]]

Retrieves the index of the first valid value.

### Returns

scalar, tuple, or None

## Examples

Support for DataFrame

```
>>> kdf = ks.DataFrame({'a': [None, 2, 3, 2],
...                     'b': [None, 2.0, 3.0, 1.0],
...                     'c': [None, 200, 400, 200]},
...                     index=['Q', 'W', 'E', 'R'])
>>> kdf
   a    b    c
Q NaN NaN NaN
W 2.0 2.0 200.0
E 3.0 3.0 400.0
R 2.0 1.0 200.0
```

```
>>> kdf.first_valid_index()
'W'
```

Support for MultiIndex columns

```
>>> kdf.columns = pd.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> kdf
```

	a	b	c
	x	y	z
Q	NaN	NaN	NaN
W	2.0	2.0	200.0
E	3.0	3.0	400.0
R	2.0	1.0	200.0

```
>>> kdf.first_valid_index()
'W'
```

### Support for Series.

```
>>> s = ks.Series([None, None, 3, 4, 5], index=[100, 200, 300, 400, 500])
>>> s
```

100	NaN
200	NaN
300	3.0
400	4.0
500	5.0

```
dtype: float64
```

```
>>> s.first_valid_index()
300
```

### Support for MultiIndex

```
>>> midx = pd.MultiIndex([['lama', 'cow', 'falcon'],
...                        ['speed', 'weight', 'length']],
...                        [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                        [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> s = ks.Series([None, None, None, None, 250, 1.5, 320, 1, 0.3], index=midx)
>>> s
```

lama	speed	NaN
	weight	NaN
	length	NaN
cow	speed	NaN
	weight	250.0
	length	1.5
falcon	speed	320.0
	weight	1.0
	length	0.3

```
dtype: float64
```

```
>>> s.first_valid_index()
('cow', 'weight')
```



**databricks.koalas.DataFrame.last\_valid\_index**

`DataFrame.last_valid_index()` → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Tuple[Union[int, float, str, bytes, decimal.Decimal, datetime.date, None], ...]]

Return index for last non-NA/null value.

**Returns**

scalar, tuple, or None

**Notes**

This API only works with PySpark >= 3.0.

**Examples**

Support for DataFrame

```
>>> kdf = ks.DataFrame({'a': [1, 2, 3, None],
...                     'b': [1.0, 2.0, 3.0, None],
...                     'c': [100, 200, 400, None]},
...                     index=['Q', 'W', 'E', 'R'])
>>> kdf
```

	a	b	c
Q	1.0	1.0	100.0
W	2.0	2.0	200.0
E	3.0	3.0	400.0
R	NaN	NaN	NaN

```
>>> kdf.last_valid_index()
'E'
```

Support for MultiIndex columns

```
>>> kdf.columns = pd.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> kdf
```

	a	b	c
	x	y	z
Q	1.0	1.0	100.0
W	2.0	2.0	200.0
E	3.0	3.0	400.0
R	NaN	NaN	NaN

```
>>> kdf.last_valid_index()
'E'
```

Support for Series.

```
>>> s = ks.Series([1, 2, 3, None, None], index=[100, 200, 300, 400, 500])
>>> s
```

100	1.0
200	2.0
300	3.0
400	NaN

(continues on next page)

(continued from previous page)

```
500      NaN
dtype: float64
```

```
>>> s.last_valid_index()
300
```

### Support for MultiIndex

```
>>> midx = pd.MultiIndex([['lama', 'cow', 'falcon'],
...                       ['speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> s = ks.Series([250, 1.5, 320, 1, 0.3, None, None, None, None], index=midx)
>>> s
lama      speed      250.0
          weight       1.5
          length     320.0
cow       speed       1.0
          weight       0.3
          length      NaN
falcon    speed      NaN
          weight      NaN
          length      NaN
dtype: float64
```

```
>>> s.last_valid_index()
('cow', 'weight')
```

## 3.4.13 Serialization / IO / Conversion

<code>DataFrame.from_records(data[, index, ...])</code>	Convert structured or record ndarray to DataFrame.
<code>DataFrame.info([verbose, buf, max_cols, ...])</code>	Print a concise summary of a DataFrame.
<code>DataFrame.to_table(name[, format, mode, ...])</code>	Write the DataFrame into a Spark table.
<code>DataFrame.to_delta(path[, mode, ...])</code>	Write the DataFrame out as a Delta Lake table.
<code>DataFrame.to_parquet(path[, mode, ...])</code>	Write the DataFrame out as a Parquet file or directory.
<code>DataFrame.to_spark_io([path, format, mode, ...])</code>	Write the DataFrame out to a Spark data source.
<code>DataFrame.to_csv([path, sep, na_rep, ...])</code>	Write object to a comma-separated values (csv) file.
<code>DataFrame.to_pandas()</code>	Return a pandas DataFrame.
<code>DataFrame.to_html([buf, columns, col_space, ...])</code>	Render a DataFrame as an HTML table.
<code>DataFrame.to_numpy()</code>	A NumPy ndarray representing the values in this DataFrame or Series.
<code>DataFrame.to_koalas([index_col])</code>	Converts the existing DataFrame into a Koalas DataFrame.
<code>DataFrame.to_spark([index_col])</code>	Spark related features.
<code>DataFrame.to_string([buf, columns, ...])</code>	Render a DataFrame to a console-friendly tabular output.
<code>DataFrame.to_json([path, compression, ...])</code>	Convert the object to a JSON string.
<code>DataFrame.to_dict([orient, into])</code>	Convert the DataFrame to a dictionary.
<code>DataFrame.to_excel(excel_writer[, ...])</code>	Write object to an Excel sheet.

continues on next page

Table 54 – continued from previous page

<code>DataFrame.to_clipboard([excel, sep])</code>	Copy object to the system clipboard.
<code>DataFrame.to_markdown([buf, mode])</code>	Print Series or DataFrame in Markdown-friendly format.
<code>DataFrame.to_records([index, column_dtypes, ...])</code>	Convert DataFrame to a NumPy record array.
<code>DataFrame.to_latex([buf, columns, ...])</code>	Render an object to a LaTeX tabular environment table.
<code>DataFrame.style</code>	Property returning a Styler object containing methods for building a styled HTML representation for the DataFrame.

**databricks.koalas.DataFrame.from\_records**

**static** `DataFrame.from_records` (*data*: `Union[numpy.array, List[tuple], dict, pandas.core.frame.DataFrame]`, *index*: `Union[str, list, numpy.array]` = `None`, *exclude*: `list` = `None`, *columns*: `list` = `None`, *coerce\_float*: `bool` = `False`, *nrows*: `int` = `None`) → `databricks.koalas.frame.DataFrame`

Convert structured or record ndarray to DataFrame.

**Parameters**

**data** [ndarray (structured dtype), list of tuples, dict, or DataFrame]

**index** [string, list of fields, array-like] Field of array to use as the index, alternately a specific set of input labels to use

**exclude** [sequence, default None] Columns or fields to exclude

**columns** [sequence, default None] Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns)

**coerce\_float** [boolean, default False] Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

**nrows** [int, default None] Number of rows to read if data is an iterator

**Returns**

**df** [DataFrame]

**Examples**

Use dict as input

```
>>> ks.DataFrame.from_records({'A': [1, 2, 3]})
  A
0  1
1  2
2  3
```

Use list of tuples as input

```
>>> ks.DataFrame.from_records([(1, 2), (3, 4)])
   0  1
0  1  2
1  3  4
```

Use NumPy array as input

```
>>> ks.DataFrame.from_records(np.eye(3))
   0  1  2
0  1.0 0.0 0.0
1  0.0 1.0 0.0
2  0.0 0.0 1.0
```

## databricks.koalas.DataFrame.info

`DataFrame.info(verbose=None, buf=None, max_cols=None, null_counts=None) → None`

Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage.

### Parameters

**verbose** [bool, optional] Whether to print the full summary.

**buf** [writable buffer, defaults to sys.stdout] Where to send the output. By default, the output is printed to sys.stdout. Pass a writable buffer if you need to further process the output.

**max\_cols** [int, optional] When to switch from the verbose to the truncated output. If the DataFrame has more than *max\_cols* columns, the truncated output is used.

**null\_counts** [bool, optional] Whether to show the non-null counts.

### Returns

**None** This method prints a summary of a DataFrame and returns None.

See also:

[`DataFrame.describe`](#) Generate descriptive statistics of DataFrame columns.

## Examples

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = ks.DataFrame(
...     {"int_col": int_values, "text_col": text_values, "float_col": float_
...     ↪values},
...     columns=['int_col', 'text_col', 'float_col'])
>>> df
   int_col text_col  float_col
0         1   alpha         0.00
1         2   beta         0.25
2         3   gamma         0.50
3         4   delta         0.75
4         5  epsilon         1.00
```

Prints information of all columns:

```
>>> df.info(verbose=True)
<class 'databricks.koalas.frame.DataFrame'>
Index: 5 entries, 0 to 4
Data columns (total 3 columns):
#   Column      Non-Null Count  Dtype
---  -
0   int_col      5 non-null      int64
1   text_col     5 non-null      object
2   float_col    5 non-null      float64
dtypes: float64(1), int64(1), object(1)
```

Prints a summary of columns count and its dtypes but not per column information:

```
>>> df.info(verbose=False)
<class 'databricks.koalas.frame.DataFrame'>
Index: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```
>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open('%s/info.txt' % path, "w",
...          encoding="utf-8") as f:
...     _ = f.write(s)
>>> with open('%s/info.txt' % path) as f:
...     f.readlines()
["<class 'databricks.koalas.frame.DataFrame'>\n",
'Index: 5 entries, 0 to 4\n',
'Data columns (total 3 columns):\n',
'#   Column      Non-Null Count  Dtype  \n',
'---  -
0   int_col      5 non-null      int64  \n',
'1   text_col     5 non-null      object \n',
'2   float_col    5 non-null      float64\n',
'dtypes: float64(1), int64(1), object(1)']
```

### databricks.koalas.DataFrame.to\_pandas

DataFrame.**to\_pandas**() → pandas.core.frame.DataFrame

Return a pandas DataFrame.

**Note:** This method should only be used if the resulting pandas DataFrame is expected to be small, as all the data is loaded into the driver's memory.

## Examples

```
>>> df = ks.DataFrame([(.2, .3), (.0, .6), (.6, .0), (.2, .1)],
...                    columns=['dogs', 'cats'])
>>> df.to_pandas()
   dogs  cats
0   0.2   0.3
1   0.0   0.6
2   0.6   0.0
3   0.2   0.1
```

## `databricks.koalas.DataFrame.to_numpy`

`DataFrame.to_numpy()` → `numpy.ndarray`

A NumPy ndarray representing the values in this DataFrame or Series.

---

**Note:** This method should only be used if the resulting NumPy ndarray is expected to be small, as all the data is loaded into the driver's memory.

---

### Returns

`numpy.ndarray`

## Examples

```
>>> ks.DataFrame({"A": [1, 2], "B": [3, 4]}).to_numpy()
array([[1, 3],
       [2, 4]])
```

With heterogeneous data, the lowest common type will have to be used.

```
>>> ks.DataFrame({"A": [1, 2], "B": [3.0, 4.5]}).to_numpy()
array([[1. , 3. ],
       [2. , 4.5]])
```

For a mix of numeric and non-numeric types, the output array will have object dtype.

```
>>> df = ks.DataFrame({"A": [1, 2], "B": [3.0, 4.5], "C": pd.date_range('2000',
↳ periods=2)})
>>> df.to_numpy()
array([[1, 3.0, Timestamp('2000-01-01 00:00:00')],
       [2, 4.5, Timestamp('2000-01-02 00:00:00')]], dtype=object)
```

For Series,

```
>>> ks.Series(['a', 'b', 'a']).to_numpy()
array(['a', 'b', 'a'], dtype=object)
```

**databricks.koalas.DataFrame.to\_koalas**

`DataFrame.to_koalas` (*index\_col*: *Union[str, List[str], None]* = *None*) → `databricks.koalas.frame.DataFrame`

Converts the existing DataFrame into a Koalas DataFrame.

This method is monkey-patched into Spark's DataFrame and can be used to convert a Spark DataFrame into a Koalas DataFrame. If running on an existing Koalas DataFrame, the method returns itself.

If a Koalas DataFrame is converted to a Spark DataFrame and then back to Koalas, it will lose the index information and the original index will be turned into a normal column.

**Parameters**

**index\_col**: str or list of str, optional, default: None Index column of table in Spark.

See also:

[\*DataFrame.to\\_spark\*](#)

**Examples**

```
>>> df = ks.DataFrame({'col1': [1, 2], 'col2': [3, 4]}, columns=['col1', 'col2'])
>>> df
   col1  col2
0     1     3
1     2     4
```

```
>>> spark_df = df.to_spark()
>>> spark_df
DataFrame[col1: bigint, col2: bigint]
```

```
>>> kdf = spark_df.to_koalas()
>>> kdf
   col1  col2
0     1     3
1     2     4
```

We can specify the index columns.

```
>>> kdf = spark_df.to_koalas(index_col='col1')
>>> kdf
      col2
col1
1         3
2         4
```

Calling `to_koalas` on a Koalas DataFrame simply returns itself.

```
>>> df.to_koalas()
   col1  col2
0     1     3
1     2     4
```

### `databricks.koalas.DataFrame.to_spark`

`DataFrame.to_spark` (*index\_col: Union[str, List[str], None] = None*) → `pyspark.sql.dataframe.DataFrame`  
Spark related features. Usually, the features here are missing in pandas but Spark has it.

### `databricks.koalas.DataFrame.to_string`

`DataFrame.to_string` (*buf=None, columns=None, col\_space=None, header=True, index=True, na\_rep='NaN', formatters=None, float\_format=None, sparsify=None, index\_names=True, justify=None, max\_rows=None, max\_cols=None, show\_dimensions=False, decimal='.', line\_width=None*) → `Optional[str]`  
Render a `DataFrame` to a console-friendly tabular output.

---

**Note:** This method should only be used if the resulting pandas object is expected to be small, as all the data is loaded into the driver's memory. If the input is large, set `max_rows` parameter.

---

#### Parameters

- buf** [StringIO-like, optional] Buffer to write to.
- columns** [sequence, optional, default None] The subset of columns to write. Writes all columns by default.
- col\_space** [int, optional] The minimum width of each column.
- header** [bool, optional] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names
- index** [bool, optional, default True] Whether to print index (row) labels.
- na\_rep** [str, optional, default 'NaN'] String representation of NAN to use.
- formatters** [list or dict of one-param. functions, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.
- float\_format** [one-parameter function, optional, default None] Formatter function to apply to columns' elements if they are floats. The result of this function must be a unicode string.
- sparsify** [bool, optional, default True] Set to False for a `DataFrame` with a hierarchical index to print every multiindex key at each row.
- index\_names** [bool, optional, default True] Prints the names of the indexes.
- justify** [str, default None] How to justify the column labels. If None uses the option from the print configuration (controlled by `set_option`), 'right' out of the box. Valid values are
- left
  - right
  - center
  - justify
  - justify-all
  - start
  - end



- inherit
- match-parent
- initial
- unset.

**max\_rows** [int, optional] Maximum number of rows to display in the console.

**max\_cols** [int, optional] Maximum number of columns to display in the console.

**show\_dimensions** [bool, default False] Display DataFrame dimensions (number of rows by number of columns).

**decimal** [str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

**line\_width** [int, optional] Width to wrap a line in characters.

### Returns

**str (or unicode, depending on data and options)** String representation of the dataframe.

### See also:

[`to\_html`](#) Convert DataFrame to HTML.

### Examples

```
>>> df = ks.DataFrame({'col1': [1, 2, 3], 'col2': [4, 5, 6]}, columns=['col1',
↳ 'col2'])
>>> print(df.to_string())
   col1  col2
0      1     4
1      2     5
2      3     6
```

```
>>> print(df.to_string(max_rows=2))
   col1  col2
0      1     4
1      2     5
```

### `databricks.koalas.DataFrame.to_dict`

`DataFrame.to_dict (orient='dict', into=<class 'dict'>) → Union[List, collections.abc.Mapping]`

Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

---

**Note:** This method should only be used if the resulting pandas DataFrame is expected to be small, as all the data is loaded into the driver's memory.

---

### Parameters

**orient** [str {'dict', 'list', 'series', 'split', 'records', 'index'}] Determines the type of the values of the dictionary.

- ‘dict’ (default) : dict like {column -> {index -> value}}
- ‘list’ : dict like {column -> [values]}
- ‘series’ : dict like {column -> Series(values)}
- ‘split’ : dict like {‘index’ -> [index], ‘columns’ -> [columns], ‘data’ -> [values]}
- ‘records’ : list like [{column -> value}, ... , {column -> value}]
- ‘index’ : dict like {index -> {column -> value}}

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

**into** [class, default dict] The collections.abc.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

### Returns

**dict, list or collections.abc.Mapping** Return a collections.abc.Mapping object representing the DataFrame. The resulting transformation depends on the *orient* parameter.

### Examples

```
>>> df = ks.DataFrame({'col1': [1, 2],
...                   'col2': [0.5, 0.75]},
...                   index=['row1', 'row2'],
...                   columns=['col1', 'col2'])
>>> df
   col1  col2
row1    1  0.50
row2    2  0.75
```

```
>>> df_dict = df.to_dict()
>>> sorted([(key, sorted(values.items())) for key, values in df_dict.items()])
[('col1', [('row1', 1), ('row2', 2)]), ('col2', [('row1', 0.5), ('row2', 0.75)])]
```

You can specify the return orientation.

```
>>> df_dict = df.to_dict('series')
>>> sorted(df_dict.items())
[('col1', row1    1
row2    2
Name: col1, dtype: int64), ('col2', row1    0.50
row2    0.75
Name: col2, dtype: float64)]
```

```
>>> df_dict = df.to_dict('split')
>>> sorted(df_dict.items())
[('columns', ['col1', 'col2']), ('data', [[1..., 0.75]]), ('index', ['row1', 'row2
→'])]
```

```
>>> df_dict = df.to_dict('records')
>>> [sorted(values.items()) for values in df_dict]
[ [('col1', 1...), ('col2', 0.5)], [('col1', 2...), ('col2', 0.75)] ]
```

```
>>> df_dict = df.to_dict('index')
>>> sorted([(key, sorted(values.items())) for key, values in df_dict.items()])
[('row1', [(('col1', 1), ('col2', 0.5))], ('row2', [(('col1', 2), ('col2', 0.75))])]
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([(('col1', OrderedDict([(('row1', 1), ('row2', 2))])), ('col2',
↳ OrderedDict([(('row1', 0.5), ('row2', 0.75))]))])]
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col...', 'col...'}), defaultdict(<class 'list'>, {
↳ 'col...', 'col...'})]
```

### `databricks.koalas.DataFrame.to_markdown`

`DataFrame.to_markdown(buf=None, mode=None) → str`  
 Print Series or DataFrame in Markdown-friendly format.

**Note:** This method should only be used if the resulting pandas object is expected to be small, as all the data is loaded into the driver's memory.

#### Parameters

**buf** [writable buffer, defaults to `sys.stdout`] Where to send the output. By default, the output is printed to `sys.stdout`. Pass a writable buffer if you need to further process the output.

**mode** [str, optional] Mode in which file is opened.

**\*\*kwargs** These parameters will be passed to *tabulate*.

#### Returns

**str** Series or DataFrame in Markdown-friendly format.

#### Notes

Requires the *tabulate* package.

#### Examples

```
>>> kser = ks.Series(["elk", "pig", "dog", "quetzal"], name="animal")
>>> print(kser.to_markdown())
|   | animal |
|---:|:-----|
| 0 | elk      |
| 1 | pig      |
| 2 | dog      |
| 3 | quetzal  |
```

```
>>> kdf = ks.DataFrame(
...     data={"animal_1": ["elk", "pig"], "animal_2": ["dog", "quetzal"]}
... )
>>> print(kdf.to_markdown())
|   | animal_1 | animal_2 |
|---|:-----|:-----|
| 0 | elk       | dog      |
| 1 | pig       | quetzal  |
```

### `databricks.koalas.DataFrame.to_records`

`DataFrame.to_records` (*index=True, column\_dtypes=None, index\_dtypes=None*) → `numpy.recarray`

Convert DataFrame to a NumPy record array.

Index will be included as the first field of the record array if requested.

---

**Note:** This method should only be used if the resulting NumPy ndarray is expected to be small, as all the data is loaded into the driver's memory.

---

#### Parameters

**index** [bool, default True] Include index in resulting record array, stored in 'index' field or using the index label, if set.

**column\_dtypes** [str, type, dict, default None] If a string or type, the data type to store all columns. If a dictionary, a mapping of column names and indices (zero-indexed) to specific data types.

**index\_dtypes** [str, type, dict, default None] If a string or type, the data type to store all index levels. If a dictionary, a mapping of index level names and indices (zero-indexed) to specific data types. This mapping is applied only if *index=True*.

#### Returns

**numpy.recarray** NumPy ndarray with the DataFrame labels as fields and each row of the DataFrame as entries.

See also:

**`DataFrame.from_records`** Convert structured or record ndarray to DataFrame.

**numpy.recarray** An ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.

### Examples

```
>>> df = ks.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
...                    index=['a', 'b'])
>>> df
   A    B
a  1  0.50
b  2  0.75
```

```
>>> df.to_records()
rec.array([(('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The index can be excluded from the record array:

```
>>> df.to_records(index=False)
rec.array([(1, 0.5 ), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])
```

Specification of dtype for columns is new in pandas 0.24.0. Data types can be specified for the columns:

```
>>> df.to_records(column_dtypes={"A": "int32"})
rec.array([(('a', 1, 0.5 ), ('b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i4'), ('B', '<f8')])
```

Specification of dtype for index is new in pandas 0.24.0. Data types can also be specified for the index:

```
>>> df.to_records(index_dtypes="<S2")
rec.array([(b'a', 1, 0.5 ), (b'b', 2, 0.75)],
          dtype=[('index', 'S2'), ('A', '<i8'), ('B', '<f8')])
```

### `databricks.koalas.DataFrame.to_latex`

`DataFrame.to_latex` (*buf=None*, *columns=None*, *col\_space=None*, *header=True*, *index=True*, *na\_rep='NaN'*, *formatters=None*, *float\_format=None*, *sparsify=None*, *index\_names=True*, *bold\_rows=False*, *column\_format=None*, *longtable=None*, *escape=None*, *encoding=None*, *decimal='.'*, *multicolumn=None*, *multicolumn\_format=None*, *multirow=None*) → Optional[str]

Render an object to a LaTeX tabular environment table.

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires `usepackage{booktabs}`.

---

**Note:** This method should only be used if the resulting pandas object is expected to be small, as all the data is loaded into the driver's memory. If the input is large, consider alternative formats.

---

#### Parameters

**buf** [file descriptor or None] Buffer to write to. If None, the output is returned as a string.

**columns** [list of label, optional] The subset of columns to write. Writes all columns by default.

**col\_space** [int, optional] The minimum width of each column.

**header** [bool or list of str, default True] Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names.

**index** [bool, default True] Write row names (index).

**na\_rep** [str, default 'NaN'] Missing data representation.

**formatters** [list of functions or dict of {str: function}, optional] Formatter functions to apply to columns' elements by position or name. The result of each function must be a unicode string. List must be of length equal to the number of columns.

**float\_format** [str, optional] Format string for floating point numbers.

**sparsify** [bool, optional] Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row. By default, the value will be read from the config module.

**index\_names** [bool, default True] Prints the names of the indexes.

**bold\_rows** [bool, default False] Make the row labels bold in the output.

**column\_format** [str, optional] The columns format as specified in LaTeX table format e.g. 'rcl' for 3 columns. By default, 'l' will be used for all columns except columns of numbers, which default to 'r'.

**longtable** [bool, optional] By default, the value will be read from the pandas config module. Use a longtable environment instead of tabular. Requires adding a `usepackage{longtable}` to your LaTeX preamble.

**escape** [bool, optional] By default, the value will be read from the pandas config module. When set to False prevents from escaping latex special characters in column names.

**encoding** [str, optional] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

**decimal** [str, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

**multicolumn** [bool, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

**multicolumn\_format** [str, default 'l'] The alignment for multicolumns, similar to column\_format The default will be read from the config module.

**multirow** [bool, default False] Use multirow to enhance MultiIndex rows. Requires adding a `usepackage{multirow}` to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

### Returns

**str or None** If buf is None, returns the resulting LaTeX format as a string. Otherwise returns None.

### See also:

[`DataFrame.to\_string`](#) Render a DataFrame to a console-friendly tabular output.

[`DataFrame.to\_html`](#) Render a DataFrame as an HTML table.

### Examples

```
>>> df = ks.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']},
...                    columns=['name', 'mask', 'weapon'])
>>> print(df.to_latex(index=False))
\begin{tabular}{lll}
\toprule
    name & mask & weapon \\
\midrule
    Raphael & red & sai \\
    Donatello & purple & bo staff \\
\bottomrule
\end{tabular}
```

**databricks.koalas.DataFrame.style****property** `DataFrame.style`

Property returning a Styler object containing methods for building a styled HTML representation for the DataFrame.

**Note:** currently it collects top 1000 rows and return its pandas *pandas.io.formats.style.Styler* instance.

**Examples**

```
>>> ks.range(1001).style
<pandas.io.formats.style.Styler object at ...>
```

**3.4.14 Spark-related**

`DataFrame.spark` provides features that does not exist in pandas but in Spark. These can be accessed by `DataFrame.spark.<function/property>`.

<code>DataFrame.spark.schema([index_col])</code>	Returns the underlying Spark schema.
<code>DataFrame.spark.print_schema([index_col])</code>	Prints out the underlying Spark schema in the tree format.
<code>DataFrame.spark.frame([index_col])</code>	Return the current DataFrame as a Spark DataFrame.
<code>DataFrame.spark.cache()</code>	Yields and caches the current DataFrame.
<code>DataFrame.spark.persist([storage_level])</code>	Yields and caches the current DataFrame with a specific <code>StorageLevel</code> .
<code>DataFrame.spark.hint(name, *parameters)</code>	Specifies some hint on the current DataFrame.
<code>DataFrame.spark.to_table(name[, format, ...])</code>	Write the DataFrame into a Spark table.
<code>DataFrame.spark.to_spark_io([path, format, ...])</code>	Write the DataFrame out to a Spark data source.
<code>DataFrame.spark.explain([extended, mode])</code>	Prints the underlying (logical and physical) Spark plans to the console for debugging purpose.
<code>DataFrame.spark.apply(func[, index_col])</code>	Applies a function that takes and returns a Spark DataFrame.
<code>DataFrame.spark.repartition(num_partitions)</code>	Returns a new DataFrame partitioned by the given partitioning expressions.
<code>DataFrame.spark.coalesce(num_partitions)</code>	Returns a new DataFrame that has exactly <i>num_partitions</i> partitions.
<code>DataFrame.spark.checkpoint([eager])</code>	Returns a checkpointed version of this DataFrame.
<code>DataFrame.spark.local_checkpoint([eager])</code>	Returns a locally checkpointed version of this DataFrame.

**databricks.koalas.DataFrame.spark.schema**

`spark.schema` (*index\_col*: Union[str, List[str], None] = None) → pyspark.sql.types.StructType  
Returns the underlying Spark schema.

**Parameters**

**index\_col**: str or list of str, optional, default: None Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**Returns**

**pyspark.sql.types.StructType** The underlying Spark schema.

**Examples**

```
>>> df = ks.DataFrame({'a': list('abc'),
...                    'b': list(range(1, 4)),
...                    'c': np.arange(3, 6).astype('i1'),
...                    'd': np.arange(4.0, 7.0, dtype='float64'),
...                    'e': [True, False, True],
...                    'f': pd.date_range('20130101', periods=3)},
...                    columns=['a', 'b', 'c', 'd', 'e', 'f'])
>>> df.spark.schema().simpleString()
'struct<a:string,b:bigint,c:tinyint,d:double,e:boolean,f:timestamp>'
>>> df.spark.schema(index_col='index').simpleString()
'struct<index:bigint,a:string,b:bigint,c:tinyint,d:double,e:boolean,f:timestamp>'
```

**databricks.koalas.DataFrame.spark.print\_schema**

`spark.print_schema` (*index\_col*: Union[str, List[str], None] = None) → None  
Prints out the underlying Spark schema in the tree format.

**Parameters**

**index\_col**: str or list of str, optional, default: None Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**Returns**

None

**Examples**

```
>>> df = ks.DataFrame({'a': list('abc'),
...                    'b': list(range(1, 4)),
...                    'c': np.arange(3, 6).astype('i1'),
...                    'd': np.arange(4.0, 7.0, dtype='float64'),
...                    'e': [True, False, True],
...                    'f': pd.date_range('20130101', periods=3)},
...                    columns=['a', 'b', 'c', 'd', 'e', 'f'])
>>> df.spark.print_schema()
root
|-- a: string (nullable = false)
```

(continues on next page)



(continued from previous page)

```

|-- b: long (nullable = false)
|-- c: byte (nullable = false)
|-- d: double (nullable = false)
|-- e: boolean (nullable = false)
|-- f: timestamp (nullable = false)
>>> df.spark.print_schema(index_col='index')
root
 |-- index: long (nullable = false)
 |-- a: string (nullable = false)
 |-- b: long (nullable = false)
 |-- c: byte (nullable = false)
 |-- d: double (nullable = false)
 |-- e: boolean (nullable = false)
 |-- f: timestamp (nullable = false)

```

### `databricks.koalas.DataFrame.spark.frame`

`spark.frame(index_col: Union[str, List[str], None] = None) → pyspark.sql.dataframe.DataFrame`

Return the current DataFrame as a Spark DataFrame. `DataFrame.spark.frame()` is an alias of `DataFrame.to_spark()`.

#### Parameters

**index\_col:** str or list of str, optional, default: None Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

See also:

`DataFrame.to_spark`

`DataFrame.to_koalas`

`DataFrame.spark.frame`

### Examples

By default, this method loses the index as below.

```

>>> df = ks.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})
>>> df.to_spark().show()
+----+----+----+
|  a|  b|  c|
+----+----+----+
|  1|  4|  7|
|  2|  5|  8|
|  3|  6|  9|
+----+----+----+

```

```

>>> df = ks.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})
>>> df.spark.frame().show()
+----+----+----+
|  a|  b|  c|
+----+----+----+
|  1|  4|  7|

```

(continues on next page)

(continued from previous page)

```
| 2| 5| 8|
| 3| 6| 9|
+---+---+---+
```

If `index_col` is set, it keeps the index column as specified.

```
>>> df.to_spark(index_col="index").show()
+-----+---+---+---+
|index| a| b| c|
+-----+---+---+---+
|    0| 1| 4| 7|
|    1| 2| 5| 8|
|    2| 3| 6| 9|
+-----+---+---+---+
```

Keeping index column is useful when you want to call some Spark APIs and convert it back to Koalas DataFrame without creating a default index, which can affect performance.

```
>>> spark_df = df.to_spark(index_col="index")
>>> spark_df = spark_df.filter("a == 2")
>>> spark_df.to_koalas(index_col="index")
   a  b  c
index
1    2  5  8
```

In case of multi-index, specify a list to `index_col`.

```
>>> new_df = df.set_index("a", append=True)
>>> new_spark_df = new_df.to_spark(index_col=["index_1", "index_2"])
>>> new_spark_df.show()
+-----+-----+---+---+
|index_1|index_2| b| c|
+-----+-----+---+---+
|      0|      1| 4| 7|
|      1|      2| 5| 8|
|      2|      3| 6| 9|
+-----+-----+---+---+
```

Likewise, can be converted to back to Koalas DataFrame.

```
>>> new_spark_df.to_koalas(
...     index_col=["index_1", "index_2"])
   b  c
index_1 index_2
0      1      4  7
1      2      5  8
2      3      6  9
```

**databricks.koalas.DataFrame.spark.cache**

`spark.cache()` → `CachedDataFrame`

Yields and caches the current `DataFrame`.

The Koalas `DataFrame` is yielded as a protected resource and its corresponding data is cached which gets uncached after execution goes of the context.

If you want to specify the `StorageLevel` manually, use `DataFrame.spark.persist()`

See also:

**`DataFrame.spark.persist`**

**Examples**

```
>>> df = ks.DataFrame([(0.2, 0.3), (0.0, 0.6), (0.6, 0.0), (0.2, 0.1)],
...                    columns=['dogs', 'cats'])
>>> df
   dogs  cats
0   0.2   0.3
1   0.0   0.6
2   0.6   0.0
3   0.2   0.1
```

```
>>> with df.spark.cache() as cached_df:
...     print(cached_df.count())
...
dogs      4
cats      4
dtype: int64
```

```
>>> df = df.spark.cache()
>>> df.to_pandas().mean(axis=1)
0    0.25
1    0.30
2    0.30
3    0.15
dtype: float64
```

To uncache the dataframe, use *unpersist* function

```
>>> df.spark.unpersist()
```

**databricks.koalas.DataFrame.spark.persist**

`spark.persist(storage_level: pyspark.storagelevel.StorageLevel = StorageLevel(True, True, False, False, 1))` → `CachedDataFrame`

Yields and caches the current `DataFrame` with a specific `StorageLevel`. If a `StorageLevel` is not given, the `MEMORY_AND_DISK` level is used by default like PySpark.

The Koalas `DataFrame` is yielded as a protected resource and its corresponding data is cached which gets uncached after execution goes of the context.

See also:

**DataFrame.spark.cache****Examples**

```
>>> import pyspark
>>> df = ks.DataFrame([(0.2, .3), (.0, .6), (.6, .0), (.2, .1)],
...                   columns=['dogs', 'cats'])
>>> df
   dogs  cats
0   0.2   0.3
1   0.0   0.6
2   0.6   0.0
3   0.2   0.1
```

Set the StorageLevel to *MEMORY\_ONLY*.

```
>>> with df.spark.persist(pyspark.StorageLevel.MEMORY_ONLY) as cached_df:
...     print(cached_df.spark.storage_level)
...     print(cached_df.count())
...
Memory Serialized 1x Replicated
dogs      4
cats      4
dtype: int64
```

Set the StorageLevel to *DISK\_ONLY*.

```
>>> with df.spark.persist(pyspark.StorageLevel.DISK_ONLY) as cached_df:
...     print(cached_df.spark.storage_level)
...     print(cached_df.count())
...
Disk Serialized 1x Replicated
dogs      4
cats      4
dtype: int64
```

If a StorageLevel is not given, it uses *MEMORY\_AND\_DISK* by default.

```
>>> with df.spark.persist() as cached_df:
...     print(cached_df.spark.storage_level)
...     print(cached_df.count())
...
Disk Memory Serialized 1x Replicated
dogs      4
cats      4
dtype: int64
```

```
>>> df = df.spark.persist()
>>> df.to_pandas().mean(axis=1)
0    0.25
1    0.30
2    0.30
3    0.15
dtype: float64
```

To uncache the dataframe, use *unpersist* function

```
>>> df.spark.unpersist()
```

### `databricks.koalas.DataFrame.spark.hint`

`spark.hint` (*name: str, \*parameters*) → `ks.DataFrame`  
 Specifies some hint on the current `DataFrame`.

#### Parameters

**name** [A name of the hint.]

**parameters** [Optional parameters.]

#### Returns

**ret** [`DataFrame` with the hint.]

See also:

**broadcast** Marks a `DataFrame` as small enough for use in broadcast joins.

### Examples

```
>>> df1 = ks.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [1, 2, 3, 5]},
...                     columns=['lkey', 'value']).set_index('lkey')
>>> df2 = ks.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [5, 6, 7, 8]},
...                     columns=['rkey', 'value']).set_index('rkey')
>>> merged = df1.merge(df2.spark.hint("broadcast"), left_index=True, right_
↳ index=True)
>>> merged.spark.explain()
== Physical Plan ==
...
...BroadcastHashJoin...
...
```

### `databricks.koalas.DataFrame.spark.to_table`

`spark.to_table` (*name: str, format: Optional[str] = None, mode: str = 'overwrite', partition\_cols: Union[str, List[str], None] = None, index\_col: Union[str, List[str], None] = None, \*\*options*) → `None`

Write the `DataFrame` into a Spark table. `DataFrame.spark.to_table()` is an alias of `DataFrame.to_table()`.

#### Parameters

**name** [str, required] Table name in Spark.

**format** [string, optional] Specifies the output data source format. Some common ones are:

- 'delta'
- 'parquet'
- 'orc'
- 'json'

- 'csv'

**mode** [str { 'append', 'overwrite', 'ignore', 'error', 'errorifexists' }, default] 'overwrite'. Specifies the behavior of the save operation when the table exists already.

- 'append': Append the new data to existing data.
- 'overwrite': Overwrite existing data.
- 'ignore': Silently ignore this operation if data already exists.
- 'error' or 'errorifexists': Throw an exception if data already exists.

**partition\_cols** [str or list of str, optional, default None] Names of partitioning columns

**index\_col**: str or list of str, optional, default: None Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**options** Additional options passed directly to Spark.

### Returns

None

See also:

`read_table`

`DataFrame.to_spark_io`

`DataFrame.spark.to_spark_io`

`DataFrame.to_parquet`

### Examples

```
>>> df = ks.DataFrame(dict(
...     date=list(pd.date_range('2012-1-1 12:00:00', periods=3, freq='M')),
...     country=['KR', 'US', 'JP'],
...     code=[1, 2, 3]), columns=['date', 'country', 'code'])
>>> df
```

	date	country	code
0	2012-01-31 12:00:00	KR	1
1	2012-02-29 12:00:00	US	2
2	2012-03-31 12:00:00	JP	3

```
>>> df.to_table('%s.my_table' % db, partition_cols='date')
```

### `databricks.koalas.DataFrame.spark.to_spark_io`

`spark.to_spark_io` (*path*: Optional[str] = None, *format*: Optional[str] = None, *mode*: str = 'overwrite', *partition\_cols*: Union[str, List[str], None] = None, *index\_col*: Union[str, List[str], None] = None, *\*\*options*) → None

Write the DataFrame out to a Spark data source. `DataFrame.spark.to_spark_io()` is an alias of `DataFrame.to_spark_io()`.

### Parameters

**path** [string, optional] Path to the data source.

**format** [string, optional] Specifies the output data source format. Some common ones are:

- 'delta'
- 'parquet'
- 'orc'
- 'json'
- 'csv'

**mode** [str {'append', 'overwrite', 'ignore', 'error', 'errorifexists'}, default] 'overwrite'. Specifies the behavior of the save operation when data already.

- 'append': Append the new data to existing data.
- 'overwrite': Overwrite existing data.
- 'ignore': Silently ignore this operation if data already exists.
- 'error' or 'errorifexists': Throw an exception if data already exists.

**partition\_cols** [str or list of str, optional] Names of partitioning columns

**index\_col: str or list of str, optional, default: None** Column names to be used in Spark to represent Koalas' index. The index name in Koalas is ignored. By default, the index is always lost.

**options** [dict] All other options passed directly into Spark's data source.

#### Returns

None

See also:

`read_spark_io`

`DataFrame.to_delta`

`DataFrame.to_parquet`

`DataFrame.to_table`

`DataFrame.to_spark_io`

`DataFrame.spark.to_spark_io`

#### Examples

```
>>> df = ks.DataFrame(dict(
...     date=list(pd.date_range('2012-1-1 12:00:00', periods=3, freq='M')),
...     country=['KR', 'US', 'JP'],
...     code=[1, 2, 3]), columns=['date', 'country', 'code'])
>>> df
```

	date	country	code
0	2012-01-31 12:00:00	KR	1
1	2012-02-29 12:00:00	US	2
2	2012-03-31 12:00:00	JP	3

```
>>> df.to_spark_io(path='%s/to_spark_io/foo.json' % path, format='json')
```

**databricks.koalas.DataFrame.spark.explain**

`spark.explain` (*extended*: *Optional[bool]* = *None*, *mode*: *Optional[str]* = *None*) → *None*  
Prints the underlying (logical and physical) Spark plans to the console for debugging purpose.

**Parameters**

**extended** [boolean, default `False`.] If `False`, prints only the physical plan.

**mode** [string, default `None`.] The expected output format of plans.

**Returns**

**None**

**Examples**

```
>>> df = ks.DataFrame({'id': range(10)})
>>> df.spark.explain()
== Physical Plan ==
...
```

```
>>> df.spark.explain(True)
== Parsed Logical Plan ==
...
== Analyzed Logical Plan ==
...
== Optimized Logical Plan ==
...
== Physical Plan ==
...
```

```
>>> df.spark.explain("extended")
== Parsed Logical Plan ==
...
== Analyzed Logical Plan ==
...
== Optimized Logical Plan ==
...
== Physical Plan ==
...
```

```
>>> df.spark.explain(mode="extended")
== Parsed Logical Plan ==
...
== Analyzed Logical Plan ==
...
== Optimized Logical Plan ==
...
== Physical Plan ==
...
```



**databricks.koalas.DataFrame.spark.apply**

`spark.apply` (*func*, *index\_col*: Union[str, List[str], None] = None) → ks.DataFrame

Applies a function that takes and returns a Spark DataFrame. It allows natively apply a Spark function and column APIs with the Spark column internally used in Series or Index.

**Note:** set *index\_col* and keep the column named as so in the output Spark DataFrame to avoid using the default index to prevent performance penalty. If you omit *index\_col*, it will use default index which is potentially expensive in general.

**Note:** it will lose column labels. This is a synonym of `func(kdf.to_spark(index_col)).to_koalas(index_col)`.

**Parameters**

**func** [function] Function to apply the function against the data by using Spark DataFrame.

**Returns**

**DataFrame**

**Raises**

**ValueError** [If the output from the function is not a Spark DataFrame.]

**Examples**

```
>>> kdf = ks.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]}, columns=["a", "b"])
>>> kdf
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> kdf.spark.apply(
...     lambda sdf: sdf.selectExpr("a + b as c", "index"), index_col="index")
...
   c
index
0   5
1   7
2   9
```

The case below ends up with using the default index, which should be avoided if possible.

```
>>> kdf.spark.apply(lambda sdf: sdf.groupby("a").count().sort("a"))
   a  count
0  1      1
1  2      1
2  3      1
```

**databricks.koalas.DataFrame.spark.repartition**

`spark.repartition(num_partitions: int) → ks.DataFrame`

Returns a new DataFrame partitioned by the given partitioning expressions. The resulting DataFrame is hash partitioned.

**Parameters**

**num\_partitions** [int] The target number of partitions.

**Returns**

**DataFrame**

**Examples**

```
>>> kdf = ks.DataFrame({"age": [5, 5, 2, 2],
...                     "name": ["Bob", "Bob", "Alice", "Alice"]}).set_index("age")
>>> kdf.sort_index()
      name
age
2    Alice
2    Alice
5     Bob
5     Bob
>>> new_kdf = kdf.spark.repartition(7)
>>> new_kdf.to_spark().rdd.getNumPartitions()
7
>>> new_kdf.sort_index()
      name
age
2    Alice
2    Alice
5     Bob
5     Bob
```

**databricks.koalas.DataFrame.spark.coalesce**

`spark.coalesce(num_partitions: int) → ks.DataFrame`

Returns a new DataFrame that has exactly *num\_partitions* partitions.

---

**Note:** This operation results in a narrow dependency, e.g. if you go from 1000 partitions to 100 partitions, there will not be a shuffle, instead each of the 100 new partitions will claim 10 of the current partitions. If a larger number of partitions is requested, it will stay at the current number of partitions. However, if you're doing a drastic coalesce, e.g. to `num_partitions = 1`, this may result in your computation taking place on fewer nodes than you like (e.g. one node in the case of `num_partitions = 1`). To avoid this, you can call `repartition()`. This will add a shuffle step, but means the current upstream partitions will be executed in parallel (per whatever the current partitioning is).

---

**Parameters**

**num\_partitions** [int] The target number of partitions.

**Returns**

## DataFrame

### Examples

```
>>> kdf = ks.DataFrame({"age": [5, 5, 2, 2],
...                     "name": ["Bob", "Bob", "Alice", "Alice"]}).set_index("age")
>>> kdf.sort_index()
      name
age
2    Alice
2    Alice
5     Bob
5     Bob
>>> new_kdf = kdf.spark.coalesce(1)
>>> new_kdf.to_spark().rdd.getNumPartitions()
1
>>> new_kdf.sort_index()
      name
age
2    Alice
2    Alice
5     Bob
5     Bob
```

### databricks.koalas.DataFrame.spark.checkpoint

`spark.checkpoint` (*eager*: *bool = True*) → `ks.DataFrame`

Returns a checkpointed version of this DataFrame.

Checkpointing can be used to truncate the logical plan of this DataFrame, which is especially useful in iterative algorithms where the plan may grow exponentially. It will be saved to files inside the checkpoint directory set with `SparkContext.setCheckpointDir`.

#### Parameters

**eager** [bool] Whether to checkpoint this DataFrame immediately

#### Returns

**DataFrame**

### Examples

```
>>> kdf = ks.DataFrame({"a": ["a", "b", "c"]})
>>> kdf
   a
0  a
1  b
2  c
>>> new_kdf = kdf.spark.checkpoint()
>>> new_kdf
   a
0  a
1  b
2  c
```

**databricks.koalas.DataFrame.spark.local\_checkpoint**

`spark.local_checkpoint (eager: bool = True) → ks.DataFrame`

Returns a locally checkpointed version of this DataFrame.

Checkpointing can be used to truncate the logical plan of this DataFrame, which is especially useful in iterative algorithms where the plan may grow exponentially. Local checkpoints are stored in the executors using the caching subsystem and therefore they are not reliable.

**Parameters**

**eager** [bool] Whether to locally checkpoint this DataFrame immediately

**Returns**

**DataFrame**

**Examples**

```
>>> kdf = ks.DataFrame({"a": ["a", "b", "c"]})
>>> kdf
   a
0  a
1  b
2  c
>>> new_kdf = kdf.spark.local_checkpoint()
>>> new_kdf
   a
0  a
1  b
2  c
```

**3.4.15 Plotting**

`DataFrame.plot` is both a callable method and a namespace attribute for specific plotting methods of the form `DataFrame.plot.<kind>`.

<code>DataFrame.plot</code>	alias of <code>databricks.koalas.plot.core.KoalasPlotAccessor</code>
<code>DataFrame.plot.area([x, y])</code>	Draw a stacked area plot.
<code>DataFrame.plot.barh([x, y])</code>	Make a horizontal bar plot.
<code>DataFrame.plot.bar([x, y])</code>	Vertical bar plot.
<code>DataFrame.plot.hist([bins])</code>	Draw one histogram of the DataFrame's columns.
<code>DataFrame.plot.line([x, y])</code>	Plot DataFrame/Series as lines.
<code>DataFrame.plot.pie(**kws)</code>	Generate a pie plot.
<code>DataFrame.plot.scatter(x, y, **kws)</code>	Create a scatter plot with varying marker point size and color.
<code>DataFrame.plot.density([bw_method, ind])</code>	Generate Kernel Density Estimate plot using Gaussian kernels.
<code>DataFrame.hist([bins])</code>	Draw one histogram of the DataFrame's columns.
<code>DataFrame.kde([bw_method, ind])</code>	Generate Kernel Density Estimate plot using Gaussian kernels.

**databricks.koalas.DataFrame.plot****databricks.koalas.DataFrame.plot**alias of `databricks.koalas.plot.core.KoalasPlotAccessor`**databricks.koalas.DataFrame.plot.area**`plot.area` (*x=None, y=None, \*\*kws*)

Draw a stacked area plot.

An area plot displays quantitative data visually. This function wraps the plotly area function.

**Parameters****x** [label or position, optional] Coordinates for the X axis. By default uses the index.**y** [label or position, optional] Column to plot. By default uses all columns.**stacked** [bool, default True] Area plots are stacked by default. Set to False to create a unstacked plot (matplotlib-only).**\*\*kws** [optional] Additional keyword arguments are documented in `DataFrame.plot()`.**Returns****plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).**Examples**

For Series

```
>>> df = ks.DataFrame({
...     'sales': [3, 2, 3, 9, 10, 6],
...     'signups': [5, 5, 6, 12, 14, 13],
...     'visits': [20, 42, 28, 62, 81, 50],
... }, index=pd.date_range(start='2018/01/01', end='2018/07/01',
...                          freq='M'))
>>> df.sales.plot.area()
```

For DataFrame

```
>>> df = ks.DataFrame({
...     'sales': [3, 2, 3, 9, 10, 6],
...     'signups': [5, 5, 6, 12, 14, 13],
...     'visits': [20, 42, 28, 62, 81, 50],
... }, index=pd.date_range(start='2018/01/01', end='2018/07/01',
...                          freq='M'))
>>> df.plot.area()
```

**databricks.koalas.DataFrame.plot.barh**

`plot.barh` (*x=None, y=None, \*\*kwargs*)

Make a horizontal bar plot.

A horizontal bar plot is a plot that presents quantitative data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

**Parameters**

**x** [label or position, default `DataFrame.index`] Column to be used for categories.

**y** [label or position, default All numeric columns in dataframe] Columns to be plotted from the DataFrame.

**\*\*kwargs** Keyword arguments to pass on to `databricks.koalas.DataFrame.plot()` or `databricks.koalas.Series.plot()`.

**Returns**

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

See also:

**plotly.express.bar** Plot a vertical bar plot using plotly.

**matplotlib.axes.Axes.bar** Plot a vertical bar plot using matplotlib.

**Examples**

For Series:

```
>>> df = ks.DataFrame({'lab': ['A', 'B', 'C'], 'val': [10, 30, 20]})
>>> df.val.plot.barh()
```

For DataFrame:

```
>>> df = ks.DataFrame({'lab': ['A', 'B', 'C'], 'val': [10, 30, 20]})
>>> df.plot.barh(x='lab', y='val')
```

Plot a whole DataFrame to a horizontal bar plot

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> df.plot.barh()
```

Plot a column of the DataFrame to a horizontal bar plot

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
```

(continues on next page)

(continued from previous page)

```
...             'lifespan': lifespan}, index=index)
>>> df.plot.barh(y='speed')
```

Plot DataFrame versus the desired column

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> df.plot.barh(x='lifespan')
```

### databricks.koalas.DataFrame.plot.bar

`plot.bar` (*x=None*, *y=None*, *\*\*kws*)

Vertical bar plot.

#### Parameters

- x** [label or position, optional] Allows plotting of one column versus another. If not specified, the index of the DataFrame is used.
- y** [label or position, optional] Allows plotting of one column versus another. If not specified, all numerical columns are used.
- \*\*kws** [optional] Additional keyword arguments are documented in `Koalas.Series.plot()` or `Koalas.DataFrame.plot()`.

#### Returns

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

### Examples

Basic plot.

For Series:

```
>>> s = ks.Series([1, 3, 2])
>>> s.plot.bar()
```

For DataFrame:

```
>>> df = ks.DataFrame({'lab': ['A', 'B', 'C'], 'val': [10, 30, 20]})
>>> df.plot.bar(x='lab', y='val')
```

Plot a whole dataframe to a bar plot. Each column is stacked with a distinct color along the horizontal axis.

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
```

(continues on next page)

(continued from previous page)

```
...         'lifespan': lifespan}, index=index)
>>> df.plot.bar()
```

Instead of stacking, the figure can be split by column with plotly APIs.

```
>>> from plotly.subplots import make_subplots
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> fig = (make_subplots(rows=2, cols=1)
...       .add_trace(df.plot.bar(y='speed').data[0], row=1, col=1)
...       .add_trace(df.plot.bar(y='speed').data[0], row=1, col=1)
...       .add_trace(df.plot.bar(y='lifespan').data[0], row=2, col=1))
>>> fig
```

Plot a single column.

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> df.plot.bar(y='speed')
```

Plot only selected categories for the DataFrame.

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = ks.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> df.plot.bar(x='lifespan')
```

## databricks.koalas.DataFrame.plot.hist

`plot.hist` (*bins=10*, *\*\*kws*)

Draw one histogram of the DataFrame's columns. A **histogram** is a representation of the distribution of data. This function calls `plotting.backend.plot()`, on each series in the DataFrame, resulting in one histogram per column.

### Parameters

**bins** [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, `bins + 1` bin edges are calculated and returned. If `bins` is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, `bins` is returned unmodified.

**\*\*kws** All other plotting keyword arguments to be passed to plotting backend.

### Returns



**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

## Examples

Basic plot.

For Series:

```
>>> s = ks.Series([1, 3, 2])
>>> s.plot.hist()
```

For DataFrame:

```
>>> df = pd.DataFrame(
...     np.random.randint(1, 7, 6000),
...     columns=['one'])
>>> df['two'] = df['one'] + np.random.randint(1, 7, 6000)
>>> df = ks.from_pandas(df)
>>> df.plot.hist(bins=12, alpha=0.5)
```

## databricks.koalas.DataFrame.plot.line

`plot.line(x=None, y=None, **kwargs)`

Plot DataFrame/Series as lines.

This function is useful to plot lines using Series's values as coordinates.

### Parameters

- x** [int or str, optional] Columns to use for the horizontal axis. Either the location or the label of the columns to be used. By default, it will use the DataFrame indices.
- y** [int, str, or list of them, optional] The values to be plotted. Either the location or the label of the columns to be used. By default, it will use the remaining DataFrame numeric columns.
- \*\*kwargs** Keyword arguments to pass on to `Series.plot()` or `DataFrame.plot()`.

### Returns

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

See also:

**plotly.express.line** Plot y versus x as lines and/or markers (plotly).

**matplotlib.pyplot.plot** Plot y versus x as lines and/or markers (matplotlib).

## Examples

Basic plot.

For Series:

```
>>> s = ks.Series([1, 3, 2])
>>> s.plot.line()
```

For DataFrame:

The following example shows the populations for some animals over the years.

```
>>> df = ks.DataFrame({'pig': [20, 18, 489, 675, 1776],
...                    'horse': [4, 25, 281, 600, 1900]},
...                    index=[1990, 1997, 2003, 2009, 2014])
>>> df.plot.line()
```

The following example shows the relationship between both populations.

```
>>> df = ks.DataFrame({'pig': [20, 18, 489, 675, 1776],
...                    'horse': [4, 25, 281, 600, 1900]},
...                    index=[1990, 1997, 2003, 2009, 2014])
>>> df.plot.line(x='pig', y='horse')
```

## databricks.koalas.DataFrame.plot.pie

`plot.pie(**kws)`

Generate a pie plot.

A pie plot is a proportional representation of the numerical data in a column. This function wraps `plotly.express.pie()` for the specified column.

### Parameters

**y** [int or label, optional] Label or position of the column to plot. If not provided, `subplots=True` argument must be passed (matplotlib-only).

**\*\*kws** Keyword arguments to pass on to `Koalas.Series.plot()`.

### Returns

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

## Examples

For Series:

```
>>> df = ks.DataFrame({'mass': [0.330, 4.87, 5.97],
...                    'radius': [2439.7, 6051.8, 6378.1]},
...                    index=['Mercury', 'Venus', 'Earth'])
>>> df.mass.plot.pie()
```

For DataFrame:

```
>>> df = ks.DataFrame({'mass': [0.330, 4.87, 5.97],
...                    'radius': [2439.7, 6051.8, 6378.1]},
...                    index=['Mercury', 'Venus', 'Earth'])
>>> df.plot.pie(y='mass')
```

### `databricks.koalas.DataFrame.plot.scatter`

`plot.scatter(x, y, **kws)`

Create a scatter plot with varying marker point size and color.

The coordinates of each point are defined by two dataframe columns and filled circles are used to represent each point. This kind of plot is useful to see complex correlations between two variables. Points could be for instance natural 2D coordinates like longitude and latitude in a map or, in general, any pair of metrics that can be plotted against each other.

#### Parameters

- x** [int or str] The column name or column position to be used as horizontal coordinates for each point.
- y** [int or str] The column name or column position to be used as vertical coordinates for each point.
- s** [scalar or array\_like, optional] (matplotlib-only).
- c** [str, int or array\_like, optional] (matplotlib-only).
- \*\*kws:** **Optional** Keyword arguments to pass on to `databricks.koalas.DataFrame.plot()`.

#### Returns

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

See also:

**plotly.express.scatter** Scatter plot using multiple input data formats (plotly).

**matplotlib.pyplot.scatter** Scatter plot using multiple input data formats (matplotlib).

### Examples

Let's see how to draw a scatter plot using coordinates from the values in a DataFrame's columns.

```
>>> df = ks.DataFrame([[5.1, 3.5, 0], [4.9, 3.0, 0], [7.0, 3.2, 1],
...                   [6.4, 3.2, 1], [5.9, 3.0, 2]],
...                   columns=['length', 'width', 'species'])
>>> df.plot.scatter(x='length', y='width')
```

And now with dark scheme:

```
>>> df = ks.DataFrame([[5.1, 3.5, 0], [4.9, 3.0, 0], [7.0, 3.2, 1],
...                   [6.4, 3.2, 1], [5.9, 3.0, 2]],
...                   columns=['length', 'width', 'species'])
>>> fig = df.plot.scatter(x='length', y='width')
>>> fig.update_layout(template="plotly_dark")
```

**databricks.koalas.DataFrame.plot.density**

`plot.density` (*bw\_method=None, ind=None, \*\*kwargs*)  
Generate Kernel Density Estimate plot using Gaussian kernels.

**Parameters**

**bw\_method** [scalar] The method used to calculate the estimator bandwidth. See `KernelDensity` in PySpark for more information.

**ind** [NumPy array or integer, optional] Evaluation points for the estimated PDF. If `None` (default), 1000 equally spaced points are used. If *ind* is a NumPy array, the KDE is evaluated at the points passed. If *ind* is an integer, *ind* number of equally spaced points are used.

**\*\*kwargs** [optional] Keyword arguments to pass on to `Koalas.Series.plot()`.

**Returns**

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

**Examples**

A scalar bandwidth should be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```
>>> s = ks.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> s.plot.kde(bw_method=0.3)
```

```
>>> s = ks.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> s.plot.kde(bw_method=3)
```

The *ind* parameter determines the evaluation points for the plot of the estimated KDF:

```
>>> s = ks.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> s.plot.kde(ind=[1, 2, 3, 4, 5], bw_method=0.3)
```

For `DataFrame`, it works in the same way as `Series`:

```
>>> df = ks.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> df.plot.kde(bw_method=0.3)
```

```
>>> df = ks.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> df.plot.kde(bw_method=3)
```

```
>>> df = ks.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> df.plot.kde(ind=[1, 2, 3, 4, 5, 6], bw_method=0.3)
```

**databricks.koalas.DataFrame.hist**`DataFrame.hist (bins=10, **kws)`

Draw one histogram of the DataFrame's columns. A [histogram](#) is a representation of the distribution of data. This function calls `plotting.backend.plot()`, on each series in the DataFrame, resulting in one histogram per column.

**Parameters**

**bins** [integer or sequence, default 10] Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

**\*\*kws** All other plotting keyword arguments to be passed to plotting backend.

**Returns**

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

**Examples**

Basic plot.

For Series:

```
>>> s = ks.Series([1, 3, 2])
>>> s.plot.hist()
```

For DataFrame:

```
>>> df = pd.DataFrame(
...     np.random.randint(1, 7, 6000),
...     columns=['one'])
>>> df['two'] = df['one'] + np.random.randint(1, 7, 6000)
>>> df = ks.from_pandas(df)
>>> df.plot.hist(bins=12, alpha=0.5)
```

**databricks.koalas.DataFrame.kde**`DataFrame.kde (bw_method=None, ind=None, **kws)`

Generate Kernel Density Estimate plot using Gaussian kernels.

**Parameters**

**bw\_method** [scalar] The method used to calculate the estimator bandwidth. See `KernelDensity` in PySpark for more information.

**ind** [NumPy array or integer, optional] Evaluation points for the estimated PDF. If `None` (default), 1000 equally spaced points are used. If `ind` is a NumPy array, the KDE is evaluated at the points passed. If `ind` is an integer, `ind` number of equally spaced points are used.

**\*\*kwargs** [optional] Keyword arguments to pass on to `Koalas.Series.plot()`.

**Returns**

**plotly.graph\_objs.Figure** Return an custom object when `backend!=plotly`. Return an ndarray when `subplots=True` (matplotlib-only).

## Examples

A scalar bandwidth should be specified. Using a small bandwidth value can lead to over-fitting, while using a large bandwidth value may result in under-fitting:

```
>>> s = ks.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> s.plot.kde(bw_method=0.3)
```

```
>>> s = ks.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> s.plot.kde(bw_method=3)
```

The *ind* parameter determines the evaluation points for the plot of the estimated KDF:

```
>>> s = ks.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> s.plot.kde(ind=[1, 2, 3, 4, 5], bw_method=0.3)
```

For DataFrame, it works in the same way as Series:

```
>>> df = ks.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> df.plot.kde(bw_method=0.3)
```

```
>>> df = ks.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> df.plot.kde(bw_method=3)
```

```
>>> df = ks.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> df.plot.kde(ind=[1, 2, 3, 4, 5, 6], bw_method=0.3)
```

### 3.4.16 Koalas-specific

`DataFrame.koalas` provides Koalas-specific features that exists only in Koalas. These can be accessed by `DataFrame.koalas.<function/property>`.

<code>DataFrame.koalas.attach_id_column(id_type, ...)</code>	Attach a column to be used as identifier of rows similar to the default index.
<code>DataFrame.koalas.apply_batch(func[, args])</code>	Apply a function that takes pandas DataFrame and outputs pandas DataFrame.
<code>DataFrame.koalas.transform_batch(func, ...)</code>	Transform chunks with a function that takes pandas DataFrame and outputs pandas DataFrame.

**databricks.koalas.DataFrame.koalas.attach\_id\_column**

`koalas.attach_id_column(id_type: str, column: Union[Any, Tuple]) → DataFrame`

Attach a column to be used as identifier of rows similar to the default index.

See also [Default Index type](#).

**Parameters**

**id\_type** [string] The id type.

- ‘sequence’ : a sequence that increases one by one.

---

**Note:** this uses Spark’s Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

- ‘distributed-sequence’ : a sequence that increases one by one, by group-by and group-map approach in a distributed manner.
- ‘distributed’ : a monotonically increasing sequence simply by using PySpark’s `monotonically_increasing_id` function in a fully distributed manner.

**column** [string or tuple of string] The column name.

**Returns**

**DataFrame** The DataFrame attached the column.

**Examples**

```
>>> df = ks.DataFrame({"x": ['a', 'b', 'c']})
>>> df.koalas.attach_id_column(id_type="sequence", column="id")
  x  id
0  a   0
1  b   1
2  c   2
```

```
>>> df.koalas.attach_id_column(id_type="distributed-sequence", column=0)
  x  0
0  a  0
1  b  1
2  c  2
```

```
>>> df.koalas.attach_id_column(id_type="distributed", column=0.0)
...
  x  0.0
0  a  ...
1  b  ...
2  c  ...
```

For multi-index columns:

```
>>> df = ks.DataFrame({"x", "y": ['a', 'b', 'c']})
>>> df.koalas.attach_id_column(id_type="sequence", column=("id-x", "id-y"))
  x id-x
```

(continues on next page)

(continued from previous page)

```

    y id-y
0   a    0
1   b    1
2   c    2

```

```

>>> df.koalas.attach_id_column(id_type="distributed-sequence", column=(0, 1.0))
    x    0
    y 1.0
0   a    0
1   b    1
2   c    2

```

### `databricks.koalas.DataFrame.koalas.apply_batch`

`koalas.apply_batch(func, args=(), **kws) → DataFrame`

Apply a function that takes pandas DataFrame and outputs pandas DataFrame. The pandas DataFrame given to the function is of a batch used internally.

See also [Transform and apply a function](#).

**Note:** the *func* is unable to access to the whole input frame. Koalas internally splits the input series into multiple batches and calls *func* with each batch multiple times. Therefore, operations such as global aggregations are impossible. See the example below.

```

>>> # This case does not return the length of whole frame but of the batch_
↳internally
... # used.
... def length(pdf) -> ks.DataFrame[int]:
...     return pd.DataFrame([len(pdf)])
...
>>> df = ks.DataFrame({'A': range(1000)})
>>> df.koalas.apply_batch(length)
    c0
0    83
1    83
2    83
...
10   83
11   83

```

**Note:** this API executes the function once to infer the type which is potentially expensive, for instance, when the dataset is created after aggregations or sorting.

To avoid this, specify return type in *func*, for instance, as below:

```

>>> def plus_one(x) -> ks.DataFrame[float, float]:
...     return x + 1

```

If the return type is specified, the output column names become *c0*, *c1*, *c2* ... *cn*. These names are positionally mapped to the returned DataFrame in *func*.

To specify the column names, you can assign them in a pandas friendly style as below:



```
>>> def plus_one(x) -> ks.DataFrame["a": float, "b": float]:
...     return x + 1
```

```
>>> pdf = pd.DataFrame({'a': [1, 2, 3], 'b': [3, 4, 5]})
>>> def plus_one(x) -> ks.DataFrame[zip(pdf.dtypes, pdf.columns)]:
...     return x + 1
```

When the given function has the return type annotated, the original index of the DataFrame will be lost and a default index will be attached to the result DataFrame. Please be careful about configuring the default index. See also [Default Index Type](#).

### Parameters

**func** [function] Function to apply to each pandas frame.

**args** [tuple] Positional arguments to pass to *func* in addition to the array/series.

**\*\*kwargs** Additional keyword arguments to pass as keywords arguments to *func*.

### Returns

**DataFrame**

See also:

**DataFrame.apply** For row/columnwise operations.

**DataFrame.applymap** For elementwise operations.

**DataFrame.aggregate** Only perform aggregating type operations.

**DataFrame.transform** Only perform transforming type operations.

**Series.koalas.transform\_batch** transform the search as each pandas chunks.

### Examples

```
>>> df = ks.DataFrame([(1, 2), (3, 4), (5, 6)], columns=['A', 'B'])
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> def query_func(pdf) -> ks.DataFrame[int, int]:
...     return pdf.query('A == 1')
>>> df.koalas.apply_batch(query_func)
   c0  c1
0    1    2
```

```
>>> def query_func(pdf) -> ks.DataFrame["A": int, "B": int]:
...     return pdf.query('A == 1')
>>> df.koalas.apply_batch(query_func)
   A  B
0  1  2
```

You can also omit the type hints so Koalas infers the return schema as below:

```
>>> df.koalas.apply_batch(lambda pdf: pdf.query('A == 1'))
   A  B
0  1  2
```

You can also specify extra arguments.

```
>>> def calculation(pdf, y, z) -> ks.DataFrame[int, int]:
...     return pdf ** y + z
>>> df.koalas.apply_batch(calculation, args=(10,), z=20)
   c0      c1
0   21    1044
1  59069 1048596
2 9765645 60466196
```

You can also use `np.ufunc` and built-in functions as input.

```
>>> df.koalas.apply_batch(np.add, args=(10,))
   A  B
0  11 12
1  13 14
2  15 16
```

```
>>> (df * -1).koalas.apply_batch(abs)
   A  B
0  1  2
1  3  4
2  5  6
```

### `databricks.koalas.DataFrame.koalas.transform_batch`

`koalas.transform_batch(func, *args, **kwargs) → Union[DataFrame, Series]`

Transform chunks with a function that takes pandas DataFrame and outputs pandas DataFrame. The pandas DataFrame given to the function is of a batch used internally. The length of each input and output should be the same.

See also [Transform and apply a function](#).

**Note:** the *func* is unable to access to the whole input frame. Koalas internally splits the input series into multiple batches and calls *func* with each batch multiple times. Therefore, operations such as global aggregations are impossible. See the example below.

```
>>> # This case does not return the length of whole frame but of the batch_
↳internally
... # used.
... def length(pdf) -> ks.DataFrame[int]:
...     return pd.DataFrame([len(pdf)] * len(pdf))
...
>>> df = ks.DataFrame({'A': range(1000)})
>>> df.koalas.transform_batch(length)
   c0
0   83
1   83
2   83
...
```

**Note:** this API executes the function once to infer the type which is potentially expensive, for instance, when the dataset is created after aggregations or sorting.

To avoid this, specify return type in `func`, for instance, as below:

```
>>> def plus_one(x) -> ks.DataFrame[float, float]:
...     return x + 1
```

If the return type is specified, the output column names become `c0`, `c1`, `c2` ... `cn`. These names are positionally mapped to the returned DataFrame in `func`.

To specify the column names, you can assign them in a pandas friendly style as below:

```
>>> def plus_one(x) -> ks.DataFrame['a': float, 'b': float]:
...     return x + 1
```

```
>>> pdf = pd.DataFrame({'a': [1, 2, 3], 'b': [3, 4, 5]})
>>> def plus_one(x) -> ks.DataFrame[zip(pdf.dtypes, pdf.columns)]:
...     return x + 1
```

When the given function returns DataFrame and has the return type annotated, the original index of the DataFrame will be lost and then a default index will be attached to the result. Please be careful about configuring the default index. See also [Default Index Type](#).

### Parameters

**func** [function] Function to transform each pandas frame.

**\*args** Positional arguments to pass to func.

**\*\*kwargs** Keyword arguments to pass to func.

### Returns

**DataFrame or Series**

See also:

**DataFrame.koalas.apply\_batch** For row/columnwise operations.

**Series.koalas.transform\_batch** transform the search as each pandas chunks.

### Examples

```
>>> df = ks.DataFrame([(1, 2), (3, 4), (5, 6)], columns=['A', 'B'])
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> def plus_one_func(pdf) -> ks.DataFrame[int, int]:
...     return pdf + 1
>>> df.koalas.transform_batch(plus_one_func)
```

(continues on next page)

(continued from previous page)

```

    c0  c1
0    2   3
1    4   5
2    6   7

```

```

>>> def plus_one_func(pdf) -> ks.DataFrame['A': int, 'B': int]:
...     return pdf + 1
>>> df.koalas.transform_batch(plus_one_func)
   A  B
0  2  3
1  4  5
2  6  7

```

```

>>> def plus_one_func(pdf) -> ks.Series[int]:
...     return pdf.B + 1
>>> df.koalas.transform_batch(plus_one_func)
0    3
1    5
2    7
dtype: int64

```

You can also omit the type hints so Koalas infers the return schema as below:

```

>>> df.koalas.transform_batch(lambda pdf: pdf + 1)
   A  B
0  2  3
1  4  5
2  6  7

```

```

>>> (df * -1).koalas.transform_batch(abs)
   A  B
0  1  2
1  3  4
2  5  6

```

Note that you should not transform the index. The index information will not change.

```

>>> df.koalas.transform_batch(lambda pdf: pdf.B + 1)
0    3
1    5
2    7
Name: B, dtype: int64

```

You can also specify extra arguments as below.

```

>>> df.koalas.transform_batch(lambda pdf, a, b, c: pdf.B + a + b + c, 1, 2, c=3)
0     8
1    10
2    12
Name: B, dtype: int64

```

## 3.5 Index objects

### 3.5.1 Index

---

*Index*


---

 Koalas Index that corresponds to pandas Index logically.
 

---

#### `databricks.koalas.Index`

**class** `databricks.koalas.Index`

Koalas Index that corresponds to pandas Index logically. This might hold Spark Column internally.

#### Parameters

**data** [array-like (1-dimensional)]

**dtype** [dtype, default None] If dtype is None, we find the dtype that best fits the data. If an actual dtype is provided, we coerce to that dtype if it's safe. Otherwise, an error will be raised.

**copy** [bool] Make a copy of input ndarray.

**name** [object] Name to be stored in the index.

**tupleize\_cols** [bool (default: True)] When True, attempt to create a MultiIndex if possible.

See also:

**MultiIndex** A multi-level, or hierarchical, Index.

**DatetimeIndex** Index of datetime64 data.

**Int64Index** A special case of *Index* with purely integer labels.

**Float64Index** A special case of *Index* with purely float labels.

#### Examples

```
>>> ks.DataFrame({'a': ['a', 'b', 'c']}, index=[1, 2, 3]).index
Int64Index([1, 2, 3], dtype='int64')
```

```
>>> ks.DataFrame({'a': [1, 2, 3]}, index=list('abc')).index
Index(['a', 'b', 'c'], dtype='object')
```

```
>>> ks.Index([1, 2, 3])
Int64Index([1, 2, 3], dtype='int64')
```

```
>>> ks.Index(list('abc'))
Index(['a', 'b', 'c'], dtype='object')
```

From a Series:

```
>>> s = ks.Series([1, 2, 3], index=[10, 20, 30])
>>> ks.Index(s)
Int64Index([1, 2, 3], dtype='int64')
```

From an Index:

```
>>> idx = ks.Index([1, 2, 3])
>>> ks.Index(idx)
Int64Index([1, 2, 3], dtype='int64')
```

`__init__`(\*args, \*\*kwargs)

Initialize self. See help(type(self)) for accurate signature.

## Methods

<code>__init__</code> (*args, **kwargs)	Initialize self.
<code>all</code> ([axis])	Return whether all elements are True.
<code>any</code> ([axis])	Return whether any element is True.
<code>append</code> (other)	Append a collection of Index options together.
<code>argmax</code> ()	Return a maximum argument indexer.
<code>argmin</code> ()	Return a minimum argument indexer.
<code>asof</code> (label)	Return the label from the index, or, if not present, the previous one.
<code>astype</code> (dtype)	Cast a Koalas object to a specified dtype <code>dtype</code> .
<code>copy</code> ([name, deep])	Make a copy of this object.
<code>delete</code> (loc)	Make new Index with passed location(-s) deleted.
<code>difference</code> (other[, sort])	Return a new Index with elements from the index that are not in <i>other</i> .
<code>drop</code> (labels)	Make new Index with passed list of labels deleted.
<code>drop_duplicates</code> ()	Return Index with duplicate values removed.
<code>droplevel</code> (level)	Return index with requested level(s) removed.
<code>dropna</code> ()	Return Index or MultiIndex without NA/NaN values
<code>equals</code> (other)	Determine if two Index objects contain the same elements.
<code>factorize</code> ([sort, na_sentinel])	Encode the object as an enumerated type or categorical variable.
<code>fillna</code> (value)	Fill NA/NaN values with the specified value.
<code>get_level_values</code> (level)	Return Index if a valid level is given.
<code>holds_integer</code> ()	Whether the type is an integer type.
<code>identical</code> (other)	Similar to equals, but check that other comparable attributes are also equal.
<code>insert</code> (loc, item)	Make new Index inserting new item at location.
<code>intersection</code> (other)	Form the intersection of two Index objects.
<code>is_boolean</code> ()	Return if the current index type is a boolean type.
<code>is_categorical</code> ()	Return if the current index type is a categorical type.
<code>is_floating</code> ()	Return if the current index type is a floating type.
<code>is_integer</code> ()	Return if the current index type is a integer type.
<code>is_interval</code> ()	Return if the current index type is an interval type.
<code>is_numeric</code> ()	Return if the current index type is a numeric type.
<code>is_object</code> ()	Return if the current index type is a object type.
<code>is_type_compatible</code> (kind)	Whether the index type is compatible with the provided type.
<code>isin</code> (values)	Check whether <i>values</i> are contained in Series or Index.
<code>isna</code> ()	Detect existing (non-missing) values.

continues on next page

Table 59 – continued from previous page

<code>isnull()</code>	Detect existing (non-missing) values.
<code>item()</code>	Return the first element of the underlying data as a python scalar.
<code>max()</code>	Return the maximum value of the Index.
<code>min()</code>	Return the minimum value of the Index.
<code>notna()</code>	Detect existing (non-missing) values.
<code>notnull()</code>	Detect existing (non-missing) values.
<code>nunique([dropna, approx, rsd])</code>	Return number of unique elements in the object.
<code>rename(name[, inplace])</code>	Alter Index or MultiIndex name.
<code>repeat(repeats)</code>	Repeat elements of a Index/MultiIndex.
<code>set_names(names[, level, inplace])</code>	Set Index or MultiIndex name.
<code>shift([periods, fill_value])</code>	Shift Series/Index by desired number of periods.
<code>sort(*args, **kwargs)</code>	Use sort_values instead.
<code>sort_values([ascending])</code>	Return a sorted copy of the index.
<code>symmetric_difference(other[, result_name, sort])</code>	Compute the symmetric difference of two Index objects.
<code>take(indices)</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>toPandas()</code>	Return a pandas Index.
<code>to_frame([index, name])</code>	Create a DataFrame with a column containing the Index.
<code>to_list()</code>	Return a list of the values.
<code>to_numpy([dtype, copy])</code>	A NumPy ndarray representing the values in this Index or MultiIndex.
<code>to_pandas()</code>	Return a pandas Index.
<code>to_series([name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.
<code>tolist()</code>	Return a list of the values.
<code>transpose()</code>	Return the transpose, For index, It will be index itself.
<code>union(other[, sort])</code>	Form the union of two Index objects.
<code>unique([level])</code>	Return unique values in the index.
<code>value_counts([normalize, sort, ascending, ...])</code>	Return a Series containing counts of unique values.
<code>view()</code>	this is defined as a copy with the same identity

### Attributes

<code>T</code>	Return the transpose, For index, It will be index itself.
<code>asi8</code>	Integer representation of the values.
<code>dtype</code>	Return the dtype object of the underlying data.
<code>empty</code>	Returns true if the current object is empty.
<code>has_duplicates</code>	If index has duplicates, return True, otherwise False.
<code>hasnans</code>	Return True if it has any missing values.
<code>inferred_type</code>	Return a string of the type inferred from the values.
<code>is_all_dates</code>	Return if all data types of the index are datetime.
<code>is_monotonic</code>	Return boolean if values in the object are monotonically increasing.

continues on next page

Table 60 – continued from previous page

<i>is_monotonic_decreasing</i>	Return boolean if values in the object are monotonically decreasing.
<i>is_monotonic_increasing</i>	Return boolean if values in the object are monotonically increasing.
<i>is_unique</i>	Return if the index has unique values.
<i>name</i>	Return name of the Index.
<i>names</i>	Return names of the Index.
<i>ndim</i>	Return an int representing the number of array dimensions.
<i>nlevels</i>	Number of levels in Index & MultiIndex.
<i>shape</i>	Return a tuple of the shape of the underlying data.
<i>size</i>	Return an int representing the number of elements in this object.
<i>spark_column</i>	Spark Column object representing the Series/Index.
<i>spark_type</i>	Returns the data type as defined by Spark, as a Spark DataType object.
<i>values</i>	Return an array representing the data in the Index.

## Properties

<i>Index.is_monotonic</i>	Return boolean if values in the object are monotonically increasing.
<i>Index.is_monotonic_increasing</i>	Return boolean if values in the object are monotonically increasing.
<i>Index.is_monotonic_decreasing</i>	Return boolean if values in the object are monotonically decreasing.
<i>Index.is_unique</i>	Return if the index has unique values.
<i>Index.has_duplicates</i>	If index has duplicates, return True, otherwise False.
<i>Index.hasnans</i>	Return True if it has any missing values.
<i>Index.dtype</i>	Return the dtype object of the underlying data.
<i>Index.inferred_type</i>	Return a string of the type inferred from the values.
<i>Index.is_all_dates</i>	Return if all data types of the index are datetime.
<i>Index.shape</i>	Return a tuple of the shape of the underlying data.
<i>Index.name</i>	Return name of the Index.
<i>Index.names</i>	Return names of the Index.
<i>Index.ndim</i>	Return an int representing the number of array dimensions.
<i>Index.size</i>	Return an int representing the number of elements in this object.
<i>Index.nlevels</i>	Number of levels in Index & MultiIndex.
<i>Index.empty</i>	Returns true if the current object is empty.
<i>Index.T</i>	Return the transpose, For index, It will be index itself.
<i>Index.values</i>	Return an array representing the data in the Index.



**databricks.koalas.Index.is\_monotonic****property** `Index.is_monotonic`

Return boolean if values in the object are monotonically increasing.

**Note:** the current implementation of `is_monotonic` requires to shuffle and aggregate multiple times to check the order locally and globally, which is potentially expensive. In case of multi-index, all data are transferred to single node which can easily cause out-of-memory error currently.

**Note:** Disable the Spark config `spark.sql.optimizer.nestedSchemaPruning.enabled` for multi-index if you're using Koalas < 1.7.0 with PySpark 3.1.1.

**Returns**

`is_monotonic` [bool]

**Examples**

```
>>> ser = ks.Series(['1/1/2018', '3/1/2018', '4/1/2018'])
>>> ser.is_monotonic
True
```

```
>>> df = ks.DataFrame({'dates': [None, '1/1/2018', '2/1/2018', '3/1/2018']})
>>> df.dates.is_monotonic
False
```

```
>>> df.index.is_monotonic
True
```

```
>>> ser = ks.Series([1])
>>> ser.is_monotonic
True
```

```
>>> ser = ks.Series([])
>>> ser.is_monotonic
True
```

```
>>> ser.rename("a").to_frame().set_index("a").index.is_monotonic
True
```

```
>>> ser = ks.Series([5, 4, 3, 2, 1], index=[1, 2, 3, 4, 5])
>>> ser.is_monotonic
False
```

```
>>> ser.index.is_monotonic
True
```

Support for MultiIndex

```
>>> midx = ks.MultiIndex.from_tuples(
... [ ('x', 'a'), ('x', 'b'), ('y', 'c'), ('y', 'd'), ('z', 'e') ])
>>> midx
MultiIndex([ ('x', 'a'),
              ('x', 'b'),
              ('y', 'c'),
              ('y', 'd'),
              ('z', 'e') ],
           )
>>> midx.is_monotonic
True
```

```
>>> midx = ks.MultiIndex.from_tuples(
... [ ('z', 'a'), ('z', 'b'), ('y', 'c'), ('y', 'd'), ('x', 'e') ])
>>> midx
MultiIndex([ ('z', 'a'),
              ('z', 'b'),
              ('y', 'c'),
              ('y', 'd'),
              ('x', 'e') ],
           )
>>> midx.is_monotonic
False
```

### **databricks.koalas.Index.is\_monotonic\_increasing**

#### **property** Index.is\_monotonic\_increasing

Return boolean if values in the object are monotonically increasing.

---

**Note:** the current implementation of `is_monotonic` requires to shuffle and aggregate multiple times to check the order locally and globally, which is potentially expensive. In case of multi-index, all data are transferred to single node which can easily cause out-of-memory error currently.

---

---

**Note:** Disable the Spark config `spark.sql.optimizer.nestedSchemaPruning.enabled` for multi-index if you're using Koalas < 1.7.0 with PySpark 3.1.1.

---

#### **Returns**

**is\_monotonic** [bool]

#### **Examples**

```
>>> ser = ks.Series(['1/1/2018', '3/1/2018', '4/1/2018'])
>>> ser.is_monotonic
True
```

```
>>> df = ks.DataFrame({'dates': [None, '1/1/2018', '2/1/2018', '3/1/2018']})
>>> df.dates.is_monotonic
False
```

```
>>> df.index.is_monotonic
True
```

```
>>> ser = ks.Series([1])
>>> ser.is_monotonic
True
```

```
>>> ser = ks.Series([])
>>> ser.is_monotonic
True
```

```
>>> ser.rename("a").to_frame().set_index("a").index.is_monotonic
True
```

```
>>> ser = ks.Series([5, 4, 3, 2, 1], index=[1, 2, 3, 4, 5])
>>> ser.is_monotonic
False
```

```
>>> ser.index.is_monotonic
True
```

### Support for MultiIndex

```
>>> midx = ks.MultiIndex.from_tuples(
... [('x', 'a'), ('x', 'b'), ('y', 'c'), ('y', 'd'), ('z', 'e')])
>>> midx
MultiIndex([('x', 'a'),
            ('x', 'b'),
            ('y', 'c'),
            ('y', 'd'),
            ('z', 'e')],
           )
>>> midx.is_monotonic
True
```

```
>>> midx = ks.MultiIndex.from_tuples(
... [('z', 'a'), ('z', 'b'), ('y', 'c'), ('y', 'd'), ('x', 'e')])
>>> midx
MultiIndex([('z', 'a'),
            ('z', 'b'),
            ('y', 'c'),
            ('y', 'd'),
            ('x', 'e')],
           )
>>> midx.is_monotonic
False
```

**databricks.koalas.Index.is\_monotonic\_decreasing****property** Index.**is\_monotonic\_decreasing**

Return boolean if values in the object are monotonically decreasing.

---

**Note:** the current implementation of `is_monotonic_decreasing` requires to shuffle and aggregate multiple times to check the order locally and globally, which is potentially expensive. In case of multi-index, all data are transferred to single node which can easily cause out-of-memory error currently.

---

---

**Note:** Disable the Spark config `spark.sql.optimizer.nestedSchemaPruning.enabled` for multi-index if you're using Koalas < 1.7.0 with PySpark 3.1.1.

---

**Returns**

**is\_monotonic** [bool]

**Examples**

```
>>> ser = ks.Series(['4/1/2018', '3/1/2018', '1/1/2018'])
>>> ser.is_monotonic_decreasing
True
```

```
>>> df = ks.DataFrame({'dates': [None, '3/1/2018', '2/1/2018', '1/1/2018']})
>>> df.dates.is_monotonic_decreasing
False
```

```
>>> df.index.is_monotonic_decreasing
False
```

```
>>> ser = ks.Series([1])
>>> ser.is_monotonic_decreasing
True
```

```
>>> ser = ks.Series([])
>>> ser.is_monotonic_decreasing
True
```

```
>>> ser.rename("a").to_frame().set_index("a").index.is_monotonic_decreasing
True
```

```
>>> ser = ks.Series([5, 4, 3, 2, 1], index=[1, 2, 3, 4, 5])
>>> ser.is_monotonic_decreasing
True
```

```
>>> ser.index.is_monotonic_decreasing
False
```

Support for MultiIndex

```
>>> midx = ks.MultiIndex.from_tuples(
... [ ('x', 'a'), ('x', 'b'), ('y', 'c'), ('y', 'd'), ('z', 'e') ])
>>> midx
MultiIndex([ ('x', 'a'),
              ('x', 'b'),
              ('y', 'c'),
              ('y', 'd'),
              ('z', 'e') ],
           )
>>> midx.is_monotonic_decreasing
False
```

```
>>> midx = ks.MultiIndex.from_tuples(
... [ ('z', 'e'), ('z', 'd'), ('y', 'c'), ('y', 'b'), ('x', 'a') ])
>>> midx
MultiIndex([ ('z', 'a'),
              ('z', 'b'),
              ('y', 'c'),
              ('y', 'd'),
              ('x', 'e') ],
           )
>>> midx.is_monotonic_decreasing
True
```

### **databricks.koalas.Index.is\_unique**

#### **property** Index.**is\_unique**

Return if the index has unique values.

#### **Examples**

```
>>> idx = ks.Index([1, 5, 7, 7])
>>> idx.is_unique
False
```

```
>>> idx = ks.Index([1, 5, 7])
>>> idx.is_unique
True
```

```
>>> idx = ks.Index(["Watermelon", "Orange", "Apple",
...                 "Watermelon"])
>>> idx.is_unique
False
```

```
>>> idx = ks.Index(["Orange", "Apple",
...                 "Watermelon"])
>>> idx.is_unique
True
```

**databricks.koalas.Index.has\_duplicates****property** Index.**has\_duplicates**

If index has duplicates, return True, otherwise False.

**Examples**

```
>>> idx = ks.Index([1, 5, 7, 7])
>>> idx.has_duplicates
True
```

```
>>> idx = ks.Index([1, 5, 7])
>>> idx.has_duplicates
False
```

```
>>> idx = ks.Index(["Watermelon", "Orange", "Apple",
...                "Watermelon"])
>>> idx.has_duplicates
True
```

```
>>> idx = ks.Index(["Orange", "Apple",
...                "Watermelon"])
>>> idx.has_duplicates
False
```

**databricks.koalas.Index.hasnans****property** Index.**hasnans**

Return True if it has any missing values. Otherwise, it returns False.

```
>>> ks.DataFrame({}, index=list('abc')).index.hasnans
False
```

```
>>> ks.Series(['a', None]).hasnans
True
```

```
>>> ks.Series([1.0, 2.0, np.nan]).hasnans
True
```

```
>>> ks.Series([1, 2, 3]).hasnans
False
```

```
>>> (ks.Series([1.0, 2.0, np.nan]) + 1).hasnans
True
```

```
>>> ks.Series([1, 2, 3]).rename("a").to_frame().set_index("a").index.hasnans
False
```

**databricks.koalas.Index.dtype****property** Index.**dtype**

Return the dtype object of the underlying data.

**Examples**

```
>>> s = ks.Series([1, 2, 3])
>>> s.dtype
dtype('int64')
```

```
>>> s = ks.Series(list('abc'))
>>> s.dtype
dtype('O')
```

```
>>> s = ks.Series(pd.date_range('20130101', periods=3))
>>> s.dtype
dtype('<M8[ns]')
```

```
>>> s.rename("a").to_frame().set_index("a").index.dtype
dtype('<M8[ns]')
```

**databricks.koalas.Index.inferred\_type****property** Index.**inferred\_type**

Return a string of the type inferred from the values.

**Examples**

```
>>> from datetime import datetime
>>> ks.Index([1, 2, 3]).inferred_type
'integer'
```

```
>>> ks.Index([1.0, 2.0, 3.0]).inferred_type
'floating'
```

```
>>> ks.Index(['a', 'b', 'c']).inferred_type
'string'
```

```
>>> ks.Index([True, False, True, False]).inferred_type
'boolean'
```

**databricks.koalas.Index.is\_all\_dates****property** Index.**is\_all\_dates**

Return if all data types of the index are datetime. remember that since Koalas does not support multiple data types in an index, so it returns True if any type of data is datetime.

**Examples**

```
>>> from datetime import datetime
```

```
>>> idx = ks.Index([datetime(2019, 1, 1, 0, 0, 0), datetime(2019, 2, 3, 0, 0, 0)])
>>> idx
DatetimeIndex(['2019-01-01', '2019-02-03'], dtype='datetime64[ns]', freq=None)
```

```
>>> idx.is_all_dates
True
```

```
>>> idx = ks.Index([datetime(2019, 1, 1, 0, 0, 0), None])
>>> idx
DatetimeIndex(['2019-01-01', 'NaT'], dtype='datetime64[ns]', freq=None)
```

```
>>> idx.is_all_dates
True
```

```
>>> idx = ks.Index([0, 1, 2])
>>> idx
Int64Index([0, 1, 2], dtype='int64')
```

```
>>> idx.is_all_dates
False
```

**databricks.koalas.Index.shape****property** Index.**shape**

Return a tuple of the shape of the underlying data.

**Examples**

```
>>> idx = ks.Index(['a', 'b', 'c'])
>>> idx
Index(['a', 'b', 'c'], dtype='object')
>>> idx.shape
(3,)
```

```
>>> midx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> midx
MultiIndex([('a', 'x'),
            ('b', 'y'),
            ('c', 'z')],
```

(continues on next page)



(continued from previous page)

```

    )
>>> midx.shape
(3, )

```

**databricks.koalas.Index.name****property** `Index.name`

Return name of the Index.

**databricks.koalas.Index.names****property** `Index.names`

Return names of the Index.

**databricks.koalas.Index.ndim****property** `Index.ndim`

Return an int representing the number of array dimensions.

Return 1 for Series / Index / MultiIndex.

**Examples**

For Series

```

>>> s = ks.Series([None, 1, 2, 3, 4], index=[4, 5, 2, 1, 8])
>>> s.ndim
1

```

For Index

```

>>> s.index.ndim
1

```

For MultiIndex

```

>>> midx = pd.MultiIndex([['lama', 'cow', 'falcon'],
...                        ['speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                        [1, 1, 1, 1, 1, 2, 1, 2, 2]])
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3], index=midx)
>>> s.index.ndim
1

```

**databricks.koalas.Index.size****property** `Index.size`

Return an int representing the number of elements in this object.

**Examples**

```
>>> df = ks.DataFrame([(.2, .3), (.0, .6), (.6, .0), (.2, .1)],
...                    columns=['dogs', 'cats'],
...                    index=list('abcd'))
>>> df.index.size
4
```

```
>>> df.set_index('dogs', append=True).index.size
4
```

**databricks.koalas.Index.nlevels****property** `Index.nlevels`

Number of levels in Index &amp; MultiIndex.

**Examples**

```
>>> kdf = ks.DataFrame({"a": [1, 2, 3]}, index=pd.Index(['a', 'b', 'c'], name="idx
↳"))
>>> kdf.index.nlevels
1
```

```
>>> kdf = ks.DataFrame({'a': [1, 2, 3]}, index=[list('abc'), list('def')])
>>> kdf.index.nlevels
2
```

**databricks.koalas.Index.empty****property** `Index.empty`

Returns true if the current object is empty. Otherwise, returns false.

```
>>> ks.range(10).id.empty
False
```

```
>>> ks.range(0).id.empty
True
```

```
>>> ks.DataFrame({}, index=list('abc')).index.empty
False
```

**databricks.koalas.Index.T****property** `Index.T`

Return the transpose, For index, It will be index itself.

**Examples**

```
>>> idx = ks.Index(['a', 'b', 'c'])
>>> idx
Index(['a', 'b', 'c'], dtype='object')
```

```
>>> idx.transpose()
Index(['a', 'b', 'c'], dtype='object')
```

For MultiIndex

```
>>> midx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> midx
MultiIndex([('a', 'x'),
            ('b', 'y'),
            ('c', 'z')],
           )
```

```
>>> midx.transpose()
MultiIndex([('a', 'x'),
            ('b', 'y'),
            ('c', 'z')],
           )
```

**databricks.koalas.Index.values****property** `Index.values`

Return an array representing the data in the Index.

**Warning:** We recommend using `Index.to_numpy()` instead.

**Note:** This method should only be used if the resulting NumPy ndarray is expected to be small, as all the data is loaded into the driver's memory.

**Returns**

**numpy.ndarray**

## Examples

```
>>> ks.Series([1, 2, 3, 4]).index.values
array([0, 1, 2, 3])
>>> ks.DataFrame({'a': ['a', 'b', 'c']}, index=[[1, 2, 3], [4, 5, 6]]).index.
↪values
array([(1, 4), (2, 5), (3, 6)], dtype=object)
```

## Modifying and computations

<i>Index.all([axis])</i>	Return whether all elements are True.
<i>Index.any([axis])</i>	Return whether any element is True.
<i>Index.argmax()</i>	Return a minimum argument indexer.
<i>Index.argmax()</i>	Return a maximum argument indexer.
<i>Index.copy([name, deep])</i>	Make a copy of this object.
<i>Index.delete(loc)</i>	Make new Index with passed location(-s) deleted.
<i>Index.equals(other)</i>	Determine if two Index objects contain the same elements.
<i>Index.factorize([sort, na_sentinel])</i>	Encode the object as an enumerated type or categorical variable.
<i>Index.identical(other)</i>	Similar to equals, but check that other comparable attributes are also equal.
<i>Index.insert(loc, item)</i>	Make new Index inserting new item at location.
<i>Index.is_boolean()</i>	Return if the current index type is a boolean type.
<i>Index.is_categorical()</i>	Return if the current index type is a categorical type.
<i>Index.is_floating()</i>	Return if the current index type is a floating type.
<i>Index.is_integer()</i>	Return if the current index type is a integer type.
<i>Index.is_interval()</i>	Return if the current index type is an interval type.
<i>Index.is_numeric()</i>	Return if the current index type is a numeric type.
<i>Index.is_object()</i>	Return if the current index type is a object type.
<i>Index.drop(labels)</i>	Make new Index with passed list of labels deleted.
<i>Index.drop_duplicates()</i>	Return Index with duplicate values removed.
<i>Index.min()</i>	Return the minimum value of the Index.
<i>Index.max()</i>	Return the maximum value of the Index.
<i>Index.rename(name[, inplace])</i>	Alter Index or MultiIndex name.
<i>Index.repeat(repeats)</i>	Repeat elements of a Index/MultiIndex.
<i>Index.take(indices)</i>	Return the elements in the given <i>positional</i> indices along an axis.
<i>Index.unique([level])</i>	Return unique values in the index.
<i>Index.nunique([dropna, approx, rsd])</i>	Return number of unique elements in the object.
<i>Index.value_counts([normalize, sort, ...])</i>	Return a Series containing counts of unique values.

**databricks.koalas.Index.all**

`Index.all (axis: Union[int, str] = 0) → bool`

Return whether all elements are True.

Returns True unless there at least one element within a series that is False or equivalent (e.g. zero or empty)

**Parameters**

**axis** [{0 or 'index'}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.

**Examples**

```
>>> ks.Series([True, True]).all()
True
```

```
>>> ks.Series([True, False]).all()
False
```

```
>>> ks.Series([0, 1]).all()
False
```

```
>>> ks.Series([1, 2, 3]).all()
True
```

```
>>> ks.Series([True, True, None]).all()
True
```

```
>>> ks.Series([True, False, None]).all()
False
```

```
>>> ks.Series([]).all()
True
```

```
>>> ks.Series([np.nan]).all()
True
```

```
>>> df = ks.Series([True, False, None]).rename("a").to_frame()
>>> df.set_index("a").index.all()
False
```

**databricks.koalas.Index.any**

`Index.any (axis: Union[int, str] = 0) → bool`

Return whether any element is True.

Returns False unless there at least one element within a series that is True or equivalent (e.g. non-zero or non-empty).

**Parameters**

**axis** [{0 or 'index'}, default 0] Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.

## Examples

```
>>> ks.Series([False, False]).any()
False
```

```
>>> ks.Series([True, False]).any()
True
```

```
>>> ks.Series([0, 0]).any()
False
```

```
>>> ks.Series([0, 1, 2]).any()
True
```

```
>>> ks.Series([False, False, None]).any()
False
```

```
>>> ks.Series([True, False, None]).any()
True
```

```
>>> ks.Series([]).any()
False
```

```
>>> ks.Series([np.nan]).any()
False
```

```
>>> df = ks.Series([True, False, None]).rename("a").to_frame()
>>> df.set_index("a").index.any()
True
```

## databricks.koalas.Index.argmax

`Index.argmax()` → int

Return a minimum argument indexer.

### Parameters

**skipna** [bool, default True]

### Returns

**minimum argument indexer**

### Examples

```
>>> kidx = ks.Index([10, 9, 8, 7, 100, 5, 4, 3, 100, 3])
>>> kidx
Int64Index([10, 9, 8, 7, 100, 5, 4, 3, 100, 3], dtype='int64')
```

```
>>> kidx.argmin()
7
```

### `databricks.koalas.Index.argmax`

`Index.argmax()` → int  
Return a maximum argument indexer.

#### Parameters

**skipna** [bool, default True]

#### Returns

**maximum argument indexer**

### Examples

```
>>> kidx = ks.Index([10, 9, 8, 7, 100, 5, 4, 3, 100, 3])
>>> kidx
Int64Index([10, 9, 8, 7, 100, 5, 4, 3, 100, 3], dtype='int64')
```

```
>>> kidx.argmax()
4
```

### `databricks.koalas.Index.copy`

`Index.copy(name=None, deep=None)` → `databricks.koalas.indexes.base.Index`  
Make a copy of this object. `name` sets those attributes on the new object.

#### Parameters

**name** [string, optional] to set name of index

**deep** [None] this parameter is not supported but just dummy parameter to match pandas.

### Examples

```
>>> df = ks.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=['cobra', 'viper', 'sidewinder'],
...                    columns=['max_speed', 'shield'])
>>> df
```

	max_speed	shield
cobra	1	2
viper	4	5
sidewinder	7	8

(continues on next page)

(continued from previous page)

```
>>> df.index
Index(['cobra', 'viper', 'sidewinder'], dtype='object')
```

**Copy index**

```
>>> df.index.copy()
Index(['cobra', 'viper', 'sidewinder'], dtype='object')
```

**Copy index with name**

```
>>> df.index.copy(name='snake')
Index(['cobra', 'viper', 'sidewinder'], dtype='object', name='snake')
```

**databricks.koalas.Index.delete**

`Index.delete` (*loc*) → `databricks.koalas.indexes.base.Index`

Make new Index with passed location(-s) deleted.

**Note:** this API can be pretty expensive since it is based on a global sequence internally.

**Returns**

**new\_index** [Index]

**Examples**

```
>>> kidx = ks.Index([10, 10, 9, 8, 4, 2, 4, 4, 2, 2, 10, 10])
>>> kidx
Int64Index([10, 10, 9, 8, 4, 2, 4, 4, 2, 2, 10, 10], dtype='int64')
```

```
>>> kidx.delete(0).sort_values()
Int64Index([2, 2, 2, 4, 4, 4, 8, 9, 10, 10, 10], dtype='int64')
```

```
>>> kidx.delete([0, 1, 2, 3, 10, 11]).sort_values()
Int64Index([2, 2, 2, 4, 4, 4], dtype='int64')
```

**MultiIndex**

```
>>> kidx = ks.MultiIndex.from_tuples([('a', 'x', 1), ('b', 'y', 2), ('c', 'z', 3)])
>>> kidx
MultiIndex([(a, x, 1),
            (b, y, 2),
            (c, z, 3)],
           )
```

```
>>> kidx.delete([0, 2]).sort_values()
MultiIndex([(b, y, 2)],
           )
```



**databricks.koalas.Index.equals**

`Index.equals` (*other*) → bool

Determine if two Index objects contain the same elements.

**Returns**

**bool** True if “other” is an Index and it has the same elements as calling index; False otherwise.

**Examples**

```
>>> from databricks.koalas.config import option_context
>>> idx = ks.Index(['a', 'b', 'c'])
>>> idx.name = "name"
>>> midx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> midx.names = ("nameA", "nameB")
```

**For Index**

```
>>> idx.equals(idx)
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     idx.equals(ks.Index(['a', 'b', 'c']))
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     idx.equals(ks.Index(['b', 'b', 'a']))
False
>>> idx.equals(midx)
False
```

**For MultiIndex**

```
>>> midx.equals(midx)
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     midx.equals(ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z
↪')]))
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     midx.equals(ks.MultiIndex.from_tuples([('c', 'z'), ('b', 'y'), ('a', 'x
↪')]))
False
>>> midx.equals(idx)
False
```

**databricks.koalas.Index.factorize**

`Index.factorize` (*sort: bool = True, na\_sentinel: Optional[int] = -1*) → Tuple[Union[Series, Index], pandas.core.indexes.base.Index]

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values.

**Parameters**

**sort** [bool, default True]

**na\_sentinel** [int or None, default -1] Value to mark “not found”. If None, will not drop the NaN from the uniques of the values.

### Returns

**codes** [Series or Index] A Series or Index that’s an indexer into *uniques*. `uniques.take(codes)` will have the same values as *values*.

**uniques** [pd.Index] The unique valid values.

---

**Note:** Even if there’s a missing value in *values*, *uniques* will *not* contain an entry for it.

---

### Examples

```
>>> kser = ks.Series(['b', None, 'a', 'c', 'b'])
>>> codes, uniques = kser.factorize()
>>> codes
0    1
1   -1
2    0
3    2
4    1
dtype: int32
>>> uniques
Index(['a', 'b', 'c'], dtype='object')
```

```
>>> codes, uniques = kser.factorize(na_sentinel=None)
>>> codes
0    1
1    3
2    0
3    2
4    1
dtype: int32
>>> uniques
Index(['a', 'b', 'c', None], dtype='object')
```

```
>>> codes, uniques = kser.factorize(na_sentinel=-2)
>>> codes
0    1
1   -2
2    0
3    2
4    1
dtype: int32
>>> uniques
Index(['a', 'b', 'c'], dtype='object')
```

For Index:

```
>>> kidx = ks.Index(['b', None, 'a', 'c', 'b'])
>>> codes, uniques = kidx.factorize()
>>> codes
```

(continues on next page)

(continued from previous page)

```
Int64Index([1, -1, 0, 2, 1], dtype='int64')
>>> uniques
Index(['a', 'b', 'c'], dtype='object')
```

**databricks.koalas.Index.identical**Index.**identical** (*other*) → bool

Similar to equals, but check that other comparable attributes are also equal.

**Returns****bool** If two Index objects have equal elements and same type True, otherwise False.**Examples**

```
>>> from databricks.koalas.config import option_context
>>> idx = ks.Index(['a', 'b', 'c'])
>>> midx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
```

**For Index**

```
>>> idx.identical(idx)
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     idx.identical(ks.Index(['a', 'b', 'c']))
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     idx.identical(ks.Index(['b', 'b', 'a']))
False
>>> idx.identical(midx)
False
```

**For MultiIndex**

```
>>> midx.identical(midx)
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     midx.identical(ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')
... ↪ 'z')]))
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     midx.identical(ks.MultiIndex.from_tuples([('c', 'z'), ('b', 'y'), ('a', 'x')
... ↪ 'z')]))
False
>>> midx.identical(idx)
False
```

**databricks.koalas.Index.insert**

`Index.insert(loc: int, item)` → `databricks.koalas.indexes.base.Index`  
Make new Index inserting new item at location.

Follows Python list.append semantics for negative values.

**Parameters**

**loc** [int]

**item** [object]

**Returns**

**new\_index** [Index]

**Examples**

```
>>> kidx = ks.Index([1, 2, 3, 4, 5])
>>> kidx.insert(3, 100)
Int64Index([1, 2, 3, 100, 4, 5], dtype='int64')
```

For negative values

```
>>> kidx = ks.Index([1, 2, 3, 4, 5])
>>> kidx.insert(-3, 100)
Int64Index([1, 2, 100, 3, 4, 5], dtype='int64')
```

**databricks.koalas.Index.is\_boolean**

`Index.is_boolean()` → bool  
Return if the current index type is a boolean type.

**Examples**

```
>>> ks.DataFrame({'a': [1]}, index=[True]).index.is_boolean()
True
```

**databricks.koalas.Index.is\_categorical**

`Index.is_categorical()` → bool  
Return if the current index type is a categorical type.

### Examples

```
>>> ks.DataFrame({'a': [1]}, index=[1]).index.is_categorical()  
False
```

### `databricks.koalas.Index.is_floating`

`Index.is_floating()` → bool

Return if the current index type is a floating type.

### Examples

```
>>> ks.DataFrame({'a': [1]}, index=[1]).index.is_floating()  
False
```

### `databricks.koalas.Index.is_integer`

`Index.is_integer()` → bool

Return if the current index type is a integer type.

### Examples

```
>>> ks.DataFrame({'a': [1]}, index=[1]).index.is_integer()  
True
```

### `databricks.koalas.Index.is_interval`

`Index.is_interval()` → bool

Return if the current index type is an interval type.

### Examples

```
>>> ks.DataFrame({'a': [1]}, index=[1]).index.is_interval()  
False
```

### `databricks.koalas.Index.is_numeric`

`Index.is_numeric()` → bool

Return if the current index type is a numeric type.

## Examples

```
>>> ks.DataFrame({'a': [1]}, index=[1]).index.is_numeric()
True
```

## `databricks.koalas.Index.is_object`

`Index.is_object()` → bool

Return if the current index type is a object type.

## Examples

```
>>> ks.DataFrame({'a': [1]}, index=["a"]).index.is_object()
True
```

## `databricks.koalas.Index.drop`

`Index.drop(labels)` → `databricks.koalas.indexes.base.Index`

Make new Index with passed list of labels deleted.

### Parameters

**labels** [array-like]

### Returns

**dropped** [Index]

## Examples

```
>>> index = ks.Index([1, 2, 3])
>>> index
Int64Index([1, 2, 3], dtype='int64')
```

```
>>> index.drop([1])
Int64Index([2, 3], dtype='int64')
```

## `databricks.koalas.Index.drop_duplicates`

`Index.drop_duplicates()` → `databricks.koalas.indexes.base.Index`

Return Index with duplicate values removed.

### Returns

**deduplicated** [Index]

See also:

[\*`Series.drop\_duplicates`\*](#) Equivalent method on Series.

[\*`DataFrame.drop\_duplicates`\*](#) Equivalent method on DataFrame.

## Examples

Generate an pandas.Index with duplicate values.

```
>>> idx = ks.Index(['lama', 'cow', 'lama', 'beetle', 'lama', 'hippo'])
```

```
>>> idx.drop_duplicates().sort_values()
Index(['beetle', 'cow', 'hippo', 'lama'], dtype='object')
```

## databricks.koalas.Index.min

`Index.min()` → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Tuple[Union[int, float, str, bytes, decimal.Decimal, datetime.date, None], ...]]  
Return the minimum value of the Index.

### Returns

**scalar** Minimum value.

**See also:**

**`Index.max`** Return the maximum value of the object.

**`Series.min`** Return the minimum value in a Series.

**`DataFrame.min`** Return the minimum values in a DataFrame.

## Examples

```
>>> idx = ks.Index([3, 2, 1])
>>> idx.min()
1
```

```
>>> idx = ks.Index(['c', 'b', 'a'])
>>> idx.min()
'a'
```

For a MultiIndex, the maximum is determined lexicographically.

```
>>> idx = ks.MultiIndex.from_tuples([('a', 'x', 1), ('b', 'y', 2)])
>>> idx.min()
('a', 'x', 1)
```

## databricks.koalas.Index.max

`Index.max()` → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Tuple[Union[int, float, str, bytes, decimal.Decimal, datetime.date, None], ...]]  
Return the maximum value of the Index.

### Returns

**scalar** Maximum value.

**See also:**

**Index.min** Return the minimum value in an Index.

**Series.max** Return the maximum value in a Series.

**DataFrame.max** Return the maximum values in a DataFrame.

## Examples

```
>>> idx = ks.Index([3, 2, 1])
>>> idx.max()
3
```

```
>>> idx = ks.Index(['c', 'b', 'a'])
>>> idx.max()
'c'
```

For a MultiIndex, the maximum is determined lexicographically.

```
>>> idx = ks.MultiIndex.from_tuples([('a', 'x', 1), ('b', 'y', 2)])
>>> idx.max()
('b', 'y', 2)
```

## databricks.koalas.Index.rename

**Index.rename** (*name*: Union[Any, Tuple, List[Union[Any, Tuple]]], *inplace*: bool = False) → Optional[databricks.koalas.indexes.base.Index]  
Alter Index or MultiIndex name. Able to set new names without level. Defaults to returning new index.

### Parameters

**name** [label or list of labels] Name(s) to set.

**inplace** [boolean, default False] Modifies the object directly, instead of creating a new Index or MultiIndex.

### Returns

**Index or MultiIndex** The same type as the caller or None if inplace is True.

## Examples

```
>>> df = ks.DataFrame({'a': ['A', 'C'], 'b': ['A', 'B']}, columns=['a', 'b'])
>>> df.index.rename("c")
Int64Index([0, 1], dtype='int64', name='c')
```

```
>>> df.set_index("a", inplace=True)
>>> df.index.rename("d")
Index(['A', 'C'], dtype='object', name='d')
```

You can also change the index name in place.

```
>>> df.index.rename("e", inplace=True)
>>> df.index
Index(['A', 'C'], dtype='object', name='e')
```



```
>>> df
      b
e
A  A
C  B
```

### Support for MultiIndex

```
>>> kidx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y')])
>>> kidx.names = ['hello', 'koalas']
>>> kidx
MultiIndex([('a', 'x'),
            ('b', 'y')],
            names=['hello', 'koalas'])
```

```
>>> kidx.rename(['aloha', 'databricks'])
MultiIndex([('a', 'x'),
            ('b', 'y')],
            names=['aloha', 'databricks'])
```

## databricks.koalas.Index.repeat

`Index.repeat (repeats: int) → databricks.koalas.indexes.base.Index`

Repeat elements of a Index/MultiIndex.

Returns a new Index/MultiIndex where each element of the current Index/MultiIndex is repeated consecutively a given number of times.

### Parameters

**repeats** [int] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty Index.

### Returns

**repeated\_index** [Index/MultiIndex] Newly created Index/MultiIndex with repeated elements.

See also:

[\*Series.repeat\*](#) Equivalent function for Series.

## Examples

```
>>> idx = ks.Index(['a', 'b', 'c'])
>>> idx
Index(['a', 'b', 'c'], dtype='object')
>>> idx.repeat(2)
Index(['a', 'b', 'c', 'a', 'b', 'c'], dtype='object')
```

For MultiIndex,

```
>>> midx = ks.MultiIndex.from_tuples([('x', 'a'), ('x', 'b'), ('y', 'c')])
>>> midx
MultiIndex([('x', 'a'),
            ('x', 'b'),
            ('y', 'c')],
            names=['x', 'y'])
```

(continues on next page)

(continued from previous page)

```

        ('y', 'c')],
    )
>>> midx.repeat(2)
MultiIndex([(('x', 'a'),
              ('x', 'b'),
              ('y', 'c'),
              ('x', 'a'),
              ('x', 'b'),
              ('y', 'c'))],
           )
>>> midx.repeat(0)
MultiIndex([], )

```

**databricks.koalas.Index.take**

`Index.take(indices) → Union[Series, Index]`

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

**Parameters**

**indices** [array-like] An array of ints indicating which positions to take.

**Returns**

**taken** [same type as caller] An array-like containing the elements taken from the object.

**See also:**

**DataFrame.loc** Select a subset of a DataFrame by labels.

**DataFrame.iloc** Select a subset of a DataFrame by positions.

**numpy.take** Take elements from an array along an axis.

**Examples**

Series

```

>>> kser = ks.Series([100, 200, 300, 400, 500])
>>> kser
0    100
1    200
2    300
3    400
4    500
dtype: int64

```

```

>>> kser.take([0, 2, 4]).sort_index()
0    100
2    300
4    500
dtype: int64

```

Index

```
>>> kidx = ks.Index([100, 200, 300, 400, 500])
>>> kidx
Int64Index([100, 200, 300, 400, 500], dtype='int64')
```

```
>>> kidx.take([0, 2, 4]).sort_values()
Int64Index([100, 300, 500], dtype='int64')
```

### MultiIndex

```
>>> kmidx = ks.MultiIndex.from_tuples([("x", "a"), ("x", "b"), ("x", "c")])
>>> kmidx
MultiIndex([('x', 'a'),
            ('x', 'b'),
            ('x', 'c')],
           )
```

```
>>> kmidx.take([0, 2])
MultiIndex([('x', 'a'),
            ('x', 'c')],
           )
```

## databricks.koalas.Index.unique

`Index.unique (level=None) → databricks.koalas.indexes.base.Index`

Return unique values in the index.

Be aware the order of unique values might be different than `pandas.Index.unique`

### Parameters

**level** [int or str, optional, default is None]

### Returns

**Index without duplicates**

See also:

[\*Series.unique\*](#)

[\*groupby.SeriesGroupBy.unique\*](#)

### Examples

```
>>> ks.DataFrame({'a': ['a', 'b', 'c']}, index=[1, 1, 3]).index.unique().sort_
↪values()
Int64Index([1, 3], dtype='int64')
```

```
>>> ks.DataFrame({'a': ['a', 'b', 'c']}, index=['d', 'e', 'e']).index.unique().
↪sort_values()
Index(['d', 'e'], dtype='object')
```

### MultiIndex

```
>>> ks.MultiIndex.from_tuples([("A", "X"), ("A", "Y"), ("A", "X")]).unique()
...
MultiIndex([('A', 'X'),
            ('A', 'Y')],
           )
```

## databricks.koalas.Index.nunique

`Index.nunique(dropna: bool = True, approx: bool = False, rsd: float = 0.05) → int`

Return number of unique elements in the object. Excludes NA values by default.

### Parameters

**dropna** [bool, default True] Don't include NaN in the count.

**approx: bool, default False** If False, will use the exact algorithm and return the exact number of unique. If True, it uses the HyperLogLog approximate algorithm, which is significantly faster for large amount of data. Note: This parameter is specific to Koalas and is not found in pandas.

**rsd: float, default 0.05** Maximum estimation error allowed in the HyperLogLog algorithm. Note: Just like approx this parameter is specific to Koalas.

### Returns

int

See also:

**DataFrame.nunique** Method nunique for DataFrame.

**Series.count** Count non-NA/null observations in the Series.

## Examples

```
>>> ks.Series([1, 2, 3, np.nan]).nunique()
3
```

```
>>> ks.Series([1, 2, 3, np.nan]).nunique(dropna=False)
4
```

On big data, we recommend using the approximate algorithm to speed up this function. The result will be very close to the exact unique count.

```
>>> ks.Series([1, 2, 3, np.nan]).nunique(approx=True)
3
```

```
>>> idx = ks.Index([1, 1, 2, None])
>>> idx
Float64Index([1.0, 1.0, 2.0, nan], dtype='float64')
```

```
>>> idx.nunique()
2
```

```
>>> idx.nunique(dropna=False)
3
```

## databricks.koalas.Index.value\_counts

`Index.value_counts` (*normalize=False, sort=True, ascending=False, bins=None, dropna=True*) → Series

Return a Series containing counts of unique values. The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

### Parameters

**normalize** [boolean, default False] If True then the object returned will contain the relative frequencies of the unique values.

**sort** [boolean, default True] Sort by values.

**ascending** [boolean, default False] Sort in ascending order.

**bins** [Not Yet Supported]

**dropna** [boolean, default True] Don't include counts of NaN.

### Returns

**counts** [Series]

### See also:

[`Series.count`](#) Number of non-NA elements in a Series.

## Examples

For Series

```
>>> df = ks.DataFrame({'x': [0, 0, 1, 1, 1, np.nan]})
>>> df.x.value_counts()
1.0    3
0.0    2
Name: x, dtype: int64
```

With *normalize* set to *True*, returns the relative frequency by dividing all values by the sum of values.

```
>>> df.x.value_counts(normalize=True)
1.0    0.6
0.0    0.4
Name: x, dtype: float64
```

**dropna** With *dropna* set to *False* we can also see NaN index values.

```
>>> df.x.value_counts(dropna=False)
1.0    3
0.0    2
NaN    1
Name: x, dtype: int64
```

For Index

```
>>> idx = ks.Index([3, 1, 2, 3, 4, np.nan])
>>> idx
Float64Index([3.0, 1.0, 2.0, 3.0, 4.0, nan], dtype='float64')
```

```
>>> idx.value_counts().sort_index()
1.0    1
2.0    1
3.0    2
4.0    1
dtype: int64
```

**sort**

With *sort* set to *False*, the result wouldn't be sorted by number of count.

```
>>> idx.value_counts(sort=True).sort_index()
1.0    1
2.0    1
3.0    2
4.0    1
dtype: int64
```

**normalize**

With *normalize* set to *True*, returns the relative frequency by dividing all values by the sum of values.

```
>>> idx.value_counts(normalize=True).sort_index()
1.0    0.2
2.0    0.2
3.0    0.4
4.0    0.2
dtype: float64
```

**dropna**

With *dropna* set to *False* we can also see NaN index values.

```
>>> idx.value_counts(dropna=False).sort_index()
1.0    1
2.0    1
3.0    2
4.0    1
NaN    1
dtype: int64
```

For MultiIndex.

```
>>> midx = pd.MultiIndex([['lama', 'cow', 'falcon'],
...                       ['speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [1, 1, 1, 1, 1, 2, 1, 2, 2]])
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3], index=midx)
>>> s.index
MultiIndex([( 'lama', 'weight'),
            ( 'lama', 'weight'),
            ( 'lama', 'weight'),
            (  'cow', 'weight'),
            (  'cow', 'weight'),
```

(continues on next page)

(continued from previous page)

```
(    'cow', 'length'),
    ('falcon', 'weight'),
    ('falcon', 'length'),
    ('falcon', 'length')],
)
```

```
>>> s.index.value_counts().sort_index()
(cow, length)      1
(cow, weight)      2
(falcon, length)   2
(falcon, weight)   1
(lama, weight)     3
dtype: int64
```

```
>>> s.index.value_counts(normalize=True).sort_index()
(cow, length)      0.111111
(cow, weight)      0.222222
(falcon, length)   0.222222
(falcon, weight)   0.111111
(lama, weight)     0.333333
dtype: float64
```

If Index has name, keep the name up.

```
>>> idx = ks.Index([0, 0, 0, 1, 1, 2, 3], name='koalas')
>>> idx.value_counts().sort_index()
0      3
1      2
2      1
3      1
Name: koalas, dtype: int64
```

## Compatibility with MultiIndex

<code>Index.set_names(names[, level, inplace])</code>	Set Index or MultiIndex name.
<code>Index.droplevel(level)</code>	Return index with requested level(s) removed.

### databricks.koalas.Index.set\_names

`Index.set_names(names, level=None, inplace=False)` → Optional[databricks.koalas.indexes.base.Index]  
Set Index or MultiIndex name. Able to set new names partially and by level.

#### Parameters

**names** [label or list of label] Name(s) to set.

**level** [int, label or list of int or label, optional] If the index is a MultiIndex, level(s) to set (None for all levels). Otherwise level must be None.

**inplace** [bool, default False] Modifies the object directly, instead of creating a new Index or MultiIndex.

#### Returns

**Index** The same type as the caller or None if inplace is True.

**See also:**

**`Index.rename`** Able to set new names without level.

## Examples

```
>>> idx = ks.Index([1, 2, 3, 4])
>>> idx
Int64Index([1, 2, 3, 4], dtype='int64')
```

```
>>> idx.set_names('quarter')
Int64Index([1, 2, 3, 4], dtype='int64', name='quarter')
```

**For MultiIndex**

```
>>> idx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y')])
>>> idx
MultiIndex([('a', 'x'),
            ('b', 'y')],
           )
```

```
>>> idx.set_names(['kind', 'year'], inplace=True)
>>> idx
MultiIndex([('a', 'x'),
            ('b', 'y')],
           names=['kind', 'year'])
```

```
>>> idx.set_names('species', level=0)
MultiIndex([('a', 'x'),
            ('b', 'y')],
           names=['species', 'year'])
```

## **databricks.koalas.Index.droplevel**

**Index.droplevel** (*level*) → databricks.koalas.indexes.base.Index

Return index with requested level(s) removed. If resulting index has only 1 level left, the result will be of Index type, not MultiIndex.

### **Parameters**

**level** [int, str, tuple, or list-like, default 0] If a string is given, must be the name of a level If list-like, elements must be names or indexes of levels.

### **Returns**

**Index or MultiIndex**



## Examples

```
>>> midx = ks.DataFrame({'a': ['a', 'b']}, index=[['a', 'x'], ['b', 'y']], [1, 2]).index
>>> midx
MultiIndex([('a', 'b', 1),
            ('x', 'y', 2)],
            )
>>> midx.droplevel([0, 1])
Int64Index([1, 2], dtype='int64')
>>> midx.droplevel(0)
MultiIndex([('b', 1),
            ('y', 2)],
            )
>>> midx.names = [("a", "b"), "b", "c"]
>>> midx.droplevel(['a', 'b'])
MultiIndex([('b', 1),
            ('y', 2)],
            names=['b', 'c'])
```

## Missing Values

<code>Index.fillna(value)</code>	Fill NA/NaN values with the specified value.
<code>Index.dropna()</code>	Return Index or MultiIndex without NA/NaN values
<code>Index.isna()</code>	Detect existing (non-missing) values.
<code>Index.notna()</code>	Detect existing (non-missing) values.

## databricks.koalas.Index.fillna

`Index.fillna(value: Union[int, float, str, bytes, decimal.Decimal, datetime.date, None])` → `databricks.koalas.indexes.base.Index`  
Fill NA/NaN values with the specified value.

### Parameters

**value** [scalar] Scalar value to use to fill holes (e.g. 0). This value cannot be a list-likes.

### Returns

**Index** : filled with value

## Examples

```
>>> ki = ks.DataFrame({'a': ['a', 'b', 'c']}, index=[1, 2, None]).index
>>> ki
Float64Index([1.0, 2.0, nan], dtype='float64')
```

```
>>> ki.fillna(0)
Float64Index([1.0, 2.0, 0.0], dtype='float64')
```

**databricks.koalas.Index.dropna**

`Index.dropna()` → `databricks.koalas.indexes.base.Index`  
 Return Index or MultiIndex without NA/NaN values

**Examples**

```
>>> df = ks.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=['cobra', 'viper', None],
...                    columns=['max_speed', 'shield'])
>>> df
```

	max_speed	shield
cobra	1	2
viper	4	5
NaN	7	8

```
>>> df.index.dropna()
Index(['cobra', 'viper'], dtype='object')
```

**Also support for MultiIndex**

```
>>> midx = pd.MultiIndex([[ 'lama', 'cow', 'falcon'],
...                       [None, 'weight', 'length']],
...                       [[0, 1, 1, 1, 1, 1, 2, 2, 2],
...                       [0, 1, 1, 0, 1, 2, 1, 1, 2]])
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, None],
...               index=midx)
>>> s
```

lama	NaN	45.0
cow	weight	200.0
	weight	1.2
	NaN	30.0
	weight	250.0
	length	1.5
falcon	weight	320.0
	weight	1.0
	length	NaN

dtype: float64

```
>>> s.index.dropna()
MultiIndex([( 'cow', 'weight'),
            ( 'cow', 'weight'),
            ( 'cow', 'weight'),
            ( 'cow', 'length'),
            ('falcon', 'weight'),
            ('falcon', 'weight'),
            ('falcon', 'length')],
           )
```

**databricks.koalas.Index.isna**

`Index.isna()` → Union[Series, Index]

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy.NaN, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

**Returns**

**Series or Index** [Mask of bool values for each element in Series] that indicates whether an element is not an NA value.

**Examples**

```
>>> ser = ks.Series([5, 6, np.NaN])
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

```
>>> ser.rename("a").to_frame().set_index("a").index.isna()
Index([False, False, True], dtype='object', name='a')
```

**databricks.koalas.Index.notna**

`Index.notna()` → Union[Series, Index]

Detect existing (non-missing) values. Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as None or numpy.NaN, get mapped to False values.

**Returns**

**Series or Index** [Mask of bool values for each element in Series] that indicates whether an element is not an NA value.

**Examples**

Show which entries in a Series are not NA.

```
>>> ser = ks.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
```

(continues on next page)

(continued from previous page)

```
2    False
dtype: bool
```

```
>>> ser.rename("a").to_frame().set_index("a").index.notna()
Index([True, True, False], dtype='object', name='a')
```

## Conversion

<code>Index.astype(dtype)</code>	Cast a Koalas object to a specified dtype <code>dtype</code> .
<code>Index.item()</code>	Return the first element of the underlying data as a python scalar.
<code>Index.to_list()</code>	Return a list of the values.
<code>Index.to_series([name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.
<code>Index.to_frame([index, name])</code>	Create a DataFrame with a column containing the Index.
<code>Index.view()</code>	this is defined as a copy with the same identity
<code>Index.to_numpy([dtype, copy])</code>	A NumPy ndarray representing the values in this Index or MultiIndex.

## databricks.koalas.Index.astype

`Index.astype(dtype: Union[str, type, numpy.dtype, pandas.core.dtypes.base.ExtensionDtype]) → Union[Index, Series]`  
 Cast a Koalas object to a specified dtype `dtype`.

### Parameters

**dtype** [data type] Use a `numpy.dtype` or Python type to cast entire pandas object to the same type.

### Returns

**casted** [same type as caller]

### See also:

[`to\_datetime`](#) Convert argument to datetime.

## Examples

```
>>> ser = ks.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
```

```
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

```
>>> ser.rename("a").to_frame().set_index("a").index.astype('int64')
Int64Index([1, 2], dtype='int64', name='a')
```

### `databricks.koalas.Index.item`

`Index.item()` → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Tuple[Union[int, float, str, bytes, decimal.Decimal, datetime.date, None], ...]]  
Return the first element of the underlying data as a python scalar.

#### Returns

**scalar** The first element of Index.

#### Raises

**ValueError** If the data is not length-1.

### Examples

```
>>> kidx = ks.Index([10])
>>> kidx.item()
10
```

### `databricks.koalas.Index.to_list`

`Index.to_list()` → List

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

---

**Note:** This method should only be used if the resulting list is expected to be small, as all the data is loaded into the driver's memory.

---

### Examples

#### Index

```
>>> idx = ks.Index([1, 2, 3, 4, 5])
>>> idx.to_list()
[1, 2, 3, 4, 5]
```

#### MultiIndex

```
>>> tuples = [(1, 'red'), (1, 'blue'), (2, 'red'), (2, 'green')]
>>> midx = ks.MultiIndex.from_tuples(tuples)
>>> midx.to_list()
[(1, 'red'), (1, 'blue'), (2, 'red'), (2, 'green')]
```

**databricks.koalas.Index.to\_series**

`Index.to_series` (*name*: `Union[Any, Tuple] = None`) → `databricks.koalas.series.Series`

Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.

**Parameters**

**name** [string, optional] name of resulting Series. If None, defaults to name of original index

**Returns**

**Series** [dtype will be based on the type of the Index values.]

**Examples**

```
>>> df = ks.DataFrame([(0.2, .3), (.0, .6), (.6, .0), (.2, .1)],
...                    columns=['dogs', 'cats'],
...                    index=list('abcd'))
>>> df['dogs'].index.to_series()
a      a
b      b
c      c
d      d
dtype: object
```

**databricks.koalas.Index.to\_frame**

`Index.to_frame` (*index*=`True`, *name*=`None`) → `databricks.koalas.frame.DataFrame`

Create a DataFrame with a column containing the Index.

**Parameters**

**index** [boolean, default True] Set the index of the returned DataFrame as the original Index.

**name** [object, default None] The passed name should substitute for the index name (if it has one).

**Returns**

**DataFrame** DataFrame containing the original Index data.

**See also:**

[`Index.to\_series`](#) Convert an Index to a Series.

[`Series.to\_frame`](#) Convert Series to DataFrame.

## Examples

```
>>> idx = ks.Index(['Ant', 'Bear', 'Cow'], name='animal')
>>> idx.to_frame()
      animal
animal
Ant      Ant
Bear     Bear
Cow      Cow
```

By default, the original Index is reused. To enforce a new Index:

```
>>> idx.to_frame(index=False)
      animal
0      Ant
1     Bear
2      Cow
```

To override the name of the resulting column, specify *name*:

```
>>> idx.to_frame(name='zoo')
      zoo
animal
Ant      Ant
Bear     Bear
Cow      Cow
```

## databricks.koalas.Index.view

`Index.view()` → `databricks.koalas.indexes.base.Index`  
 this is defined as a copy with the same identity

## databricks.koalas.Index.to\_numpy

`Index.to_numpy(dtype=None, copy=False)` → `numpy.ndarray`  
 A NumPy ndarray representing the values in this Index or MultiIndex.

---

**Note:** This method should only be used if the resulting NumPy ndarray is expected to be small, as all the data is loaded into the driver's memory.

---

### Parameters

**dtype** [str or `numpy.dtype`, optional] The dtype to pass to `numpy.asarray()`

**copy** [bool, default False] Whether to ensure that the returned value is a not a view on another array. Note that `copy=False` does not *ensure* that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.

### Returns

`numpy.ndarray`

## Examples

```
>>> ks.Series([1, 2, 3, 4]).index.to_numpy()
array([0, 1, 2, 3])
>>> ks.DataFrame({'a': ['a', 'b', 'c']}, index=[[1, 2, 3], [4, 5, 6]]).index.to_
↳numpy()
array([(1, 4), (2, 5), (3, 6)], dtype=object)
```

### 3.5.2 Spark-related

`Index.spark` provides features that does not exist in pandas but in Spark. These can be accessed by `Index.spark.<function/property>`.

<code>Index.spark.data_type</code>	Returns the data type as defined by Spark, as a Spark <code>DataType</code> object.
<code>Index.spark.column</code>	Spark Column object representing the Series/Index.
<code>Index.spark.transform(func)</code>	Applies a function that takes and returns a Spark column.

#### `databricks.koalas.Index.spark.data_type`

##### **property** `spark.data_type`

Returns the data type as defined by Spark, as a Spark `DataType` object.

#### `databricks.koalas.Index.spark.column`

##### **property** `spark.column`

Spark Column object representing the Series/Index.

---

**Note:** This Spark Column object is strictly stick to its base DataFrame the Series/Index was derived from.

---

#### `databricks.koalas.Index.spark.transform`

`spark.transform(func) → ks.Index`

Applies a function that takes and returns a Spark column. It allows to natively apply a Spark function and column APIs with the Spark column internally used in Series or Index. The output length of the Spark column should be same as input's.

---

**Note:** It requires to have the same input and output length; therefore, the aggregate Spark functions such as count does not work.

---

#### **Parameters**

**func** [function] Function to use for transforming the data by using Spark columns.

#### **Returns**

**Series or Index**



**Raises****ValueError** [If the output from the function is not a Spark column.]**Examples**

```
>>> from pyspark.sql.functions import log
>>> df = ks.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]}, columns=["a", "b"])
>>> df
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.a.spark.transform(lambda c: log(c))
0    0.000000
1    0.693147
2    1.098612
Name: a, dtype: float64
```

```
>>> df.index.spark.transform(lambda c: c + 10)
Int64Index([10, 11, 12], dtype='int64')
```

```
>>> df.a.spark.transform(lambda c: c + df.b.spark.column)
0    5
1    7
2    9
Name: a, dtype: int64
```

**Sorting**


---

<code>Index.sort_values([ascending])</code>	Return a sorted copy of the index.
---	------------------------------------

---

**databricks.koalas.Index.sort\_values**

`Index.sort_values (ascending=True)` → `databricks.koalas.indexes.base.Index`  
 Return a sorted copy of the index.

---

**Note:** This method is not supported for pandas when index has NaN value. pandas raises unexpected `TypeError`, but we support treating NaN as the smallest value.

---

**Parameters****ascending** [bool, default True] Should the index values be sorted in an ascending order.**Returns****sorted\_index** [ks.Index or ks.MultiIndex] Sorted copy of the index.**See also:**

*Series.sort\_values* Sort values of a Series.

*DataFrame.sort\_values* Sort values in a DataFrame.

## Examples

```
>>> idx = ks.Index([10, 100, 1, 1000])
>>> idx
Int64Index([10, 100, 1, 1000], dtype='int64')
```

Sort values in ascending order (default behavior).

```
>>> idx.sort_values()
Int64Index([1, 10, 100, 1000], dtype='int64')
```

Sort values in descending order.

```
>>> idx.sort_values(ascending=False)
Int64Index([1000, 100, 10, 1], dtype='int64')
```

Support for MultiIndex.

```
>>> kidx = ks.MultiIndex.from_tuples([('a', 'x', 1), ('c', 'y', 2), ('b', 'z', 3)])
>>> kidx
MultiIndex([('a', 'x', 1),
            ('c', 'y', 2),
            ('b', 'z', 3)],
           )
```

```
>>> kidx.sort_values()
MultiIndex([('b', 'z', 3),
            ('c', 'y', 2)],
           )
```

```
>>> kidx.sort_values(ascending=False)
MultiIndex([('c', 'y', 2),
            ('b', 'z', 3),
            ('a', 'x', 1)],
           )
```

## Time-specific operations

---

*Index.shift*([periods, fill\_value])

Shift Series/Index by desired number of periods.

---

**databricks.koalas.Index.shift**

`Index.shift` (*periods=1, fill\_value=None*) → Union[Series, Index]

Shift Series/Index by desired number of periods.

---

**Note:** the current implementation of shift uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

**Parameters**

**periods** [int] Number of periods to shift. Can be positive or negative.

**fill\_value** [object, optional] The scalar value to use for newly introduced missing values. The default depends on the dtype of self. For numeric data, np.nan is used.

**Returns**

**Copy of input Series/Index, shifted.**

**Examples**

```
>>> df = ks.DataFrame({'Col1': [10, 20, 15, 30, 45],
...                     'Col2': [13, 23, 18, 33, 48],
...                     'Col3': [17, 27, 22, 37, 52]},
...                   columns=['Col1', 'Col2', 'Col3'])
```

```
>>> df.Col1.shift(periods=3)
0      NaN
1      NaN
2      NaN
3     10.0
4     20.0
Name: Col1, dtype: float64
```

```
>>> df.Col2.shift(periods=3, fill_value=0)
0      0
1      0
2      0
3     13
4     23
Name: Col2, dtype: int64
```

```
>>> df.index.shift(periods=3, fill_value=0)
Int64Index([0, 0, 0, 0, 1], dtype='int64')
```

## Combining / joining / set operations

<code>Index.append(other)</code>	Append a collection of Index options together.
<code>Index.intersection(other)</code>	Form the intersection of two Index objects.
<code>Index.union(other[, sort])</code>	Form the union of two Index objects.
<code>Index.difference(other[, sort])</code>	Return a new Index with elements from the index that are not in <i>other</i> .
<code>Index.symmetric_difference(other[, ...])</code>	Compute the symmetric difference of two Index objects.

### databricks.koalas.Index.append

`Index.append(other: databricks.koalas.indexes.base.Index) → databricks.koalas.indexes.base.Index`  
 Append a collection of Index options together.

#### Parameters

**other** [Index]

#### Returns

**appended** [Index]

### Examples

```
>>> kidx = ks.Index([10, 5, 0, 5, 10, 5, 0, 10])
>>> kidx
Int64Index([10, 5, 0, 5, 10, 5, 0, 10], dtype='int64')
```

```
>>> kidx.append(kidx)
Int64Index([10, 5, 0, 5, 10, 5, 0, 10, 10, 5, 0, 5, 10, 5, 0, 10], dtype='int64')
```

#### Support for MultiIndex

```
>>> kidx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y')])
>>> kidx
MultiIndex([('a', 'x'),
            ('b', 'y')],
           )
```

```
>>> kidx.append(kidx)
MultiIndex([('a', 'x'),
            ('b', 'y'),
            ('a', 'x'),
            ('b', 'y')],
           )
```

**databricks.koalas.Index.intersection**

`Index.intersection(other)` → `databricks.koalas.indexes.base.Index`

Form the intersection of two Index objects.

This returns a new Index with elements common to the index and *other*.

**Parameters**

**other** [Index or array-like]

**Returns**

**intersection** [Index]

**Examples**

```
>>> idx1 = ks.Index([1, 2, 3, 4])
>>> idx2 = ks.Index([3, 4, 5, 6])
>>> idx1.intersection(idx2).sort_values()
Int64Index([3, 4], dtype='int64')
```

**databricks.koalas.Index.union**

`Index.union(other, sort=None)` → `databricks.koalas.indexes.base.Index`

Form the union of two Index objects.

**Parameters**

**other** [Index or array-like]

**sort** [bool or None, default None] Whether to sort the resulting Index.

**Returns**

**union** [Index]

**Examples**

Index

```
>>> idx1 = ks.Index([1, 2, 3, 4])
>>> idx2 = ks.Index([3, 4, 5, 6])
>>> idx1.union(idx2).sort_values()
Int64Index([1, 2, 3, 4, 5, 6], dtype='int64')
```

MultiIndex

```
>>> midx1 = ks.MultiIndex.from_tuples([("x", "a"), ("x", "b"), ("x", "c"), ("x",
↪ "d")])
>>> midx2 = ks.MultiIndex.from_tuples([("x", "c"), ("x", "d"), ("x", "e"), ("x",
↪ "f")])
>>> midx1.union(midx2).sort_values()
MultiIndex([('x', 'a'),
            ('x', 'b'),
            ('x', 'c'),
```

(continues on next page)

(continued from previous page)

```

        ('x', 'd'),
        ('x', 'e'),
        ('x', 'f')],
    )

```

## databricks.koalas.Index.difference

`Index.difference(other, sort=None) → databricks.koalas.indexes.base.Index`

Return a new Index with elements from the index that are not in *other*.

This is the set difference of two Index objects.

### Parameters

**other** [Index or array-like]

**sort** [True or None, default None] Whether to sort the resulting index. \* True : Attempt to sort the result. \* None : Do not sort the result.

### Returns

**difference** [Index]

## Examples

```

>>> idx1 = ks.Index([2, 1, 3, 4])
>>> idx2 = ks.Index([3, 4, 5, 6])
>>> idx1.difference(idx2, sort=True)
Int64Index([1, 2], dtype='int64')

```

## MultiIndex

```

>>> midx1 = ks.MultiIndex.from_tuples([('a', 'x', 1), ('b', 'y', 2), ('c', 'z', 3)])
>>> midx2 = ks.MultiIndex.from_tuples([('a', 'x', 1), ('b', 'z', 2), ('k', 'z', 3)])
>>> midx1.difference(midx2)
MultiIndex([('b', 'y', 2),
           ('c', 'z', 3)],
           )

```

## databricks.koalas.Index.symmetric\_difference

`Index.symmetric_difference(other, result_name=None, sort=None) →`

`databricks.koalas.indexes.base.Index`

Compute the symmetric difference of two Index objects.

### Parameters

**other** [Index or array-like]

**result\_name** [str]

**sort** [True or None, default None] Whether to sort the resulting index. \* True : Attempt to sort the result. \* None : Do not sort the result.

**Returns****symmetric\_difference** [Index]**Notes**

`symmetric_difference` contains elements that appear in either `idx1` or `idx2` but not both. Equivalent to the Index created by `idx1.difference(idx2) | idx2.difference(idx1)` with duplicates dropped.

**Examples**

```
>>> s1 = ks.Series([1, 2, 3, 4], index=[1, 2, 3, 4])
>>> s2 = ks.Series([1, 2, 3, 4], index=[2, 3, 4, 5])
```

```
>>> s1.index.symmetric_difference(s2.index)
Int64Index([5, 1], dtype='int64')
```

You can set name of result Index.

```
>>> s1.index.symmetric_difference(s2.index, result_name='koalas')
Int64Index([5, 1], dtype='int64', name='koalas')
```

You can set sort to *True*, if you want to sort the resulting index.

```
>>> s1.index.symmetric_difference(s2.index, sort=True)
Int64Index([1, 5], dtype='int64')
```

You can also use the `^` operator:

```
>>> s1.index ^ s2.index
Int64Index([5, 1], dtype='int64')
```

**Selecting**

<code>Index.asof(label)</code>	Return the label from the index, or, if not present, the previous one.
<code>Index.isin(values)</code>	Check whether <i>values</i> are contained in Series or Index.

**databricks.koalas.Index.asof**

`Index.asof(label) → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None]`

Return the label from the index, or, if not present, the previous one.

Assuming that the index is sorted, return the passed index label if it is in the index, or return the previous index label if the passed one is not in the index.

**Note:** This API is dependent on `Index.is_monotonic_increasing()` which can be expensive.

**Parameters**

**label** [object] The label up to which the method returns the latest index label.

**Returns**

**object** The passed label if it is in the index. The previous label if the passed label is not in the sorted index or *NaN* if there is no such label.

**Examples**

*Index.asof* returns the latest index label up to the passed label.

```
>>> idx = ks.Index(['2013-12-31', '2014-01-02', '2014-01-03'])
>>> idx.asof('2014-01-01')
'2013-12-31'
```

If the label is in the index, the method returns the passed label.

```
>>> idx.asof('2014-01-02')
'2014-01-02'
```

If all of the labels in the index are later than the passed label, NaN is returned.

```
>>> idx.asof('1999-01-02')
nan
```

**databricks.koalas.Index.isin**

`Index.isin(values) → Union[Series, Index]`

Check whether *values* are contained in Series or Index.

Return a boolean Series or Index showing whether each element in the Series matches an element in the passed sequence of *values* exactly.

**Parameters**

**values** [set or list-like] The sequence of values to test.

**Returns**

**isin** [Series (bool dtype) or Index (bool dtype)]

**Examples**

```
>>> s = ks.Series(['lama', 'cow', 'lama', 'beetle', 'lama',
...               'hippo'], name='animal')
>>> s.isin(['cow', 'lama'])
0      True
1      True
2      True
3     False
4      True
5     False
Name: animal, dtype: bool
```

Passing a single string as `s.isin('lama')` will raise an error. Use a list of one element instead:



```
>>> s.isin(['lama'])
0      True
1     False
2      True
3     False
4      True
5     False
Name: animal, dtype: bool
```

```
>>> s.rename("a").to_frame().set_index("a").index.isin(['lama'])
Index([True, False, True, False, True, False], dtype='object', name='a')
```

### 3.5.3 Numeric Index

<i>Int64Index</i>	Immutable sequence used for indexing and alignment.
<i>Float64Index</i>	Immutable sequence used for indexing and alignment.

#### **databricks.koalas.Int64Index**

**class** databricks.koalas.Int64Index

Immutable sequence used for indexing and alignment. The basic object storing axis labels for all pandas objects. Int64Index is a special case of *Index* with purely integer labels.

#### **Parameters**

**data** [array-like (1-dimensional)]

**dtype** [NumPy dtype (default: int64)]

**copy** [bool] Make a copy of input ndarray.

**name** [object] Name to be stored in the index.

#### **See also:**

*Index* The base Koalas Index type.

*Float64Index* A special case of *Index* with purely float labels.

#### **Notes**

An Index instance can **only** contain hashable objects.

## Examples

```
>>> ks.Int64Index([1, 2, 3])
Int64Index([1, 2, 3], dtype='int64')
```

From a Series:

```
>>> s = ks.Series([1, 2, 3], index=[10, 20, 30])
>>> ks.Int64Index(s)
Int64Index([1, 2, 3], dtype='int64')
```

From an Index:

```
>>> idx = ks.Index([1, 2, 3])
>>> ks.Int64Index(idx)
Int64Index([1, 2, 3], dtype='int64')
```

**\_\_init\_\_** (\*args, \*\*kwargs)  
Initialize self. See help(type(self)) for accurate signature.

## Methods

<b>__init__</b> (*args, **kwargs)	Initialize self.
<b>all</b> ([axis])	Return whether all elements are True.
<b>any</b> ([axis])	Return whether any element is True.
<b>append</b> (other)	Append a collection of Index options together.
<b>argmax</b> ()	Return a maximum argument indexer.
<b>argmin</b> ()	Return a minimum argument indexer.
<b>asof</b> (label)	Return the label from the index, or, if not present, the previous one.
<b>astype</b> (dtype)	Cast a Koalas object to a specified dtype <i>dtype</i> .
<b>copy</b> ([name, deep])	Make a copy of this object.
<b>delete</b> (loc)	Make new Index with passed location(-s) deleted.
<b>difference</b> (other[, sort])	Return a new Index with elements from the index that are not in <i>other</i> .
<b>drop</b> (labels)	Make new Index with passed list of labels deleted.
<b>drop_duplicates</b> ()	Return Index with duplicate values removed.
<b>droplevel</b> (level)	Return index with requested level(s) removed.
<b>dropna</b> ()	Return Index or MultiIndex without NA/NaN values
<b>equals</b> (other)	Determine if two Index objects contain the same elements.
<b>factorize</b> ([sort, na_sentinel])	Encode the object as an enumerated type or categorical variable.
<b>fillna</b> (value)	Fill NA/NaN values with the specified value.
<b>get_level_values</b> (level)	Return Index if a valid level is given.
<b>holds_integer</b> ()	Whether the type is an integer type.
<b>identical</b> (other)	Similar to equals, but check that other comparable attributes are also equal.
<b>insert</b> (loc, item)	Make new Index inserting new item at location.
<b>intersection</b> (other)	Form the intersection of two Index objects.
<b>is_boolean</b> ()	Return if the current index type is a boolean type.

continues on next page

Table 72 – continued from previous page

<code>is_categorical()</code>	Return if the current index type is a categorical type.
<code>is_floating()</code>	Return if the current index type is a floating type.
<code>is_integer()</code>	Return if the current index type is a integer type.
<code>is_interval()</code>	Return if the current index type is an interval type.
<code>is_numeric()</code>	Return if the current index type is a numeric type.
<code>is_object()</code>	Return if the current index type is a object type.
<code>is_type_compatible(kind)</code>	Whether the index type is compatible with the provided type.
<code>isin(values)</code>	Check whether <i>values</i> are contained in Series or Index.
<code>isna()</code>	Detect existing (non-missing) values.
<code>isnull()</code>	Detect existing (non-missing) values.
<code>item()</code>	Return the first element of the underlying data as a python scalar.
<code>max()</code>	Return the maximum value of the Index.
<code>min()</code>	Return the minimum value of the Index.
<code>notna()</code>	Detect existing (non-missing) values.
<code>notnull()</code>	Detect existing (non-missing) values.
<code>nunique([dropna, approx, rsd])</code>	Return number of unique elements in the object.
<code>rename(name[, inplace])</code>	Alter Index or MultiIndex name.
<code>repeat(repeats)</code>	Repeat elements of a Index/MultiIndex.
<code>set_names(names[, level, inplace])</code>	Set Index or MultiIndex name.
<code>shift([periods, fill_value])</code>	Shift Series/Index by desired number of periods.
<code>sort(*args, **kwargs)</code>	Use <code>sort_values</code> instead.
<code>sort_values([ascending])</code>	Return a sorted copy of the index.
<code>symmetric_difference(other[, result_name, sort])</code>	Compute the symmetric difference of two Index objects.
<code>take(indices)</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>toPandas()</code>	Return a pandas Index.
<code>to_frame([index, name])</code>	Create a DataFrame with a column containing the Index.
<code>to_list()</code>	Return a list of the values.
<code>to_numpy([dtype, copy])</code>	A NumPy ndarray representing the values in this Index or MultiIndex.
<code>to_pandas()</code>	Return a pandas Index.
<code>to_series([name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.
<code>tolist()</code>	Return a list of the values.
<code>transpose()</code>	Return the transpose, For index, It will be index itself.
<code>union(other[, sort])</code>	Form the union of two Index objects.
<code>unique([level])</code>	Return unique values in the index.
<code>value_counts([normalize, sort, ascending, ...])</code>	Return a Series containing counts of unique values.
<code>view()</code>	this is defined as a copy with the same identity

**Attributes**

<code>T</code>	Return the transpose, For index, It will be index itself.
<code>asi8</code>	Integer representation of the values.
<code>dtype</code>	Return the dtype object of the underlying data.
<code>empty</code>	Returns true if the current object is empty.
<code>has_duplicates</code>	If index has duplicates, return True, otherwise False.
<code>hasnans</code>	Return True if it has any missing values.
<code>inferred_type</code>	Return a string of the type inferred from the values.
<code>is_all_dates</code>	Return if all data types of the index are datetime.
<code>is_monotonic</code>	Return boolean if values in the object are monotonically increasing.
<code>is_monotonic_decreasing</code>	Return boolean if values in the object are monotonically decreasing.
<code>is_monotonic_increasing</code>	Return boolean if values in the object are monotonically increasing.
<code>is_unique</code>	Return if the index has unique values.
<code>name</code>	Return name of the Index.
<code>names</code>	Return names of the Index.
<code>ndim</code>	Return an int representing the number of array dimensions.
<code>nlevels</code>	Number of levels in Index & MultiIndex.
<code>shape</code>	Return a tuple of the shape of the underlying data.
<code>size</code>	Return an int representing the number of elements in this object.
<code>spark_column</code>	Spark Column object representing the Series/Index.
<code>spark_type</code>	Returns the data type as defined by Spark, as a Spark DataType object.
<code>values</code>	Return an array representing the data in the Index.

**databricks.koalas.Float64Index****class databricks.koalas.Float64Index**

Immutable sequence used for indexing and alignment. The basic object storing axis labels for all pandas objects. Float64Index is a special case of *Index* with purely float labels.

**Parameters**

**data** [array-like (1-dimensional)]  
**dtype** [NumPy dtype (default: float64)]  
**copy** [bool] Make a copy of input ndarray.  
**name** [object] Name to be stored in the index.

**See also:**

**Index** The base Koalas Index type.

**Int64Index** A special case of *Index* with purely integer labels.

## Notes

An Index instance can **only** contain hashable objects.

## Examples

```
>>> ks.Float64Index([1.0, 2.0, 3.0])
Float64Index([1.0, 2.0, 3.0], dtype='float64')
```

From a Series:

```
>>> s = ks.Series([1, 2, 3], index=[10, 20, 30])
>>> ks.Float64Index(s)
Float64Index([1.0, 2.0, 3.0], dtype='float64')
```

From an Index:

```
>>> idx = ks.Index([1, 2, 3])
>>> ks.Float64Index(idx)
Float64Index([1.0, 2.0, 3.0], dtype='float64')
```

**\_\_init\_\_** (\*args, \*\*kwargs)  
Initialize self. See help(type(self)) for accurate signature.

## Methods

<code>__init__</code> (*args, **kwargs)	Initialize self.
<code>all</code> ([axis])	Return whether all elements are True.
<code>any</code> ([axis])	Return whether any element is True.
<code>append</code> (other)	Append a collection of Index options together.
<code>argmax</code> ()	Return a maximum argument indexer.
<code>argmin</code> ()	Return a minimum argument indexer.
<code>asof</code> (label)	Return the label from the index, or, if not present, the previous one.
<code>astype</code> (dtype)	Cast a Koalas object to a specified dtype <code>dtype</code> .
<code>copy</code> ([name, deep])	Make a copy of this object.
<code>delete</code> (loc)	Make new Index with passed location(-s) deleted.
<code>difference</code> (other[, sort])	Return a new Index with elements from the index that are not in <i>other</i> .
<code>drop</code> (labels)	Make new Index with passed list of labels deleted.
<code>drop_duplicates</code> ()	Return Index with duplicate values removed.
<code>droplevel</code> (level)	Return index with requested level(s) removed.
<code>dropna</code> ()	Return Index or MultiIndex without NA/NaN values
<code>equals</code> (other)	Determine if two Index objects contain the same elements.
<code>factorize</code> ([sort, na_sentinel])	Encode the object as an enumerated type or categorical variable.
<code>fillna</code> (value)	Fill NA/NaN values with the specified value.
<code>get_level_values</code> (level)	Return Index if a valid level is given.
<code>holds_integer</code> ()	Whether the type is an integer type.

continues on next page

Table 74 – continued from previous page

<code>identical(other)</code>	Similar to equals, but check that other comparable attributes are also equal.
<code>insert(loc, item)</code>	Make new Index inserting new item at location.
<code>intersection(other)</code>	Form the intersection of two Index objects.
<code>is_boolean()</code>	Return if the current index type is a boolean type.
<code>is_categorical()</code>	Return if the current index type is a categorical type.
<code>is_floating()</code>	Return if the current index type is a floating type.
<code>is_integer()</code>	Return if the current index type is a integer type.
<code>is_interval()</code>	Return if the current index type is an interval type.
<code>is_numeric()</code>	Return if the current index type is a numeric type.
<code>is_object()</code>	Return if the current index type is a object type.
<code>is_type_compatible(kind)</code>	Whether the index type is compatible with the provided type.
<code>isin(values)</code>	Check whether <i>values</i> are contained in Series or Index.
<code>isna()</code>	Detect existing (non-missing) values.
<code>isnull()</code>	Detect existing (non-missing) values.
<code>item()</code>	Return the first element of the underlying data as a python scalar.
<code>max()</code>	Return the maximum value of the Index.
<code>min()</code>	Return the minimum value of the Index.
<code>notna()</code>	Detect existing (non-missing) values.
<code>notnull()</code>	Detect existing (non-missing) values.
<code>nunique([dropna, approx, rsd])</code>	Return number of unique elements in the object.
<code>rename(name[, inplace])</code>	Alter Index or MultiIndex name.
<code>repeat(repeats)</code>	Repeat elements of a Index/MultiIndex.
<code>set_names(names[, level, inplace])</code>	Set Index or MultiIndex name.
<code>shift([periods, fill_value])</code>	Shift Series/Index by desired number of periods.
<code>sort(*args, **kwargs)</code>	Use sort_values instead.
<code>sort_values([ascending])</code>	Return a sorted copy of the index.
<code>symmetric_difference(other[, result_name, sort])</code>	Compute the symmetric difference of two Index objects.
<code>take(indices)</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>toPandas()</code>	Return a pandas Index.
<code>to_frame([index, name])</code>	Create a DataFrame with a column containing the Index.
<code>to_list()</code>	Return a list of the values.
<code>to_numpy([dtype, copy])</code>	A NumPy ndarray representing the values in this Index or MultiIndex.
<code>to_pandas()</code>	Return a pandas Index.
<code>to_series([name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.
<code>tolist()</code>	Return a list of the values.
<code>transpose()</code>	Return the transpose, For index, It will be index itself.
<code>union(other[, sort])</code>	Form the union of two Index objects.
<code>unique([level])</code>	Return unique values in the index.
<code>value_counts([normalize, sort, ascending, ...])</code>	Return a Series containing counts of unique values.
<code>view()</code>	this is defined as a copy with the same identity

**Attributes**

<code>T</code>	Return the transpose, For index, It will be index itself.
<code>asi8</code>	Integer representation of the values.
<code>dtype</code>	Return the dtype object of the underlying data.
<code>empty</code>	Returns true if the current object is empty.
<code>has_duplicates</code>	If index has duplicates, return True, otherwise False.
<code>hasnans</code>	Return True if it has any missing values.
<code>inferred_type</code>	Return a string of the type inferred from the values.
<code>is_all_dates</code>	Return if all data types of the index are datetime.
<code>is_monotonic</code>	Return boolean if values in the object are monotonically increasing.
<code>is_monotonic_decreasing</code>	Return boolean if values in the object are monotonically decreasing.
<code>is_monotonic_increasing</code>	Return boolean if values in the object are monotonically increasing.
<code>is_unique</code>	Return if the index has unique values.
<code>name</code>	Return name of the Index.
<code>names</code>	Return names of the Index.
<code>ndim</code>	Return an int representing the number of array dimensions.
<code>nlevels</code>	Number of levels in Index & MultiIndex.
<code>shape</code>	Return a tuple of the shape of the underlying data.
<code>size</code>	Return an int representing the number of elements in this object.
<code>spark_column</code>	Spark Column object representing the Series/Index.
<code>spark_type</code>	Returns the data type as defined by Spark, as a Spark DataType object.
<code>values</code>	Return an array representing the data in the Index.

**3.5.4 CategoricalIndex**

<i>CategoricalIndex</i>	Index based on an underlying <i>Categorical</i> .
-------------------------	---

**databricks.koalas.CategoricalIndex**

**class** databricks.koalas.CategoricalIndex

Index based on an underlying *Categorical*.

CategoricalIndex can only take on a limited, and usually fixed, number of possible values (*categories*). Also, it might have an order, but numerical operations (additions, divisions, ...) are not possible.

**Parameters**

**data** [array-like (1-dimensional)] The values of the categorical. If *categories* are given, values not in *categories* will be replaced with NaN.

**categories** [index-like, optional] The categories for the categorical. Items need to be unique. If the categories are not given here (and also not in *dtype*), they will be inferred from the *data*.

**ordered** [bool, optional] Whether or not this categorical is treated as an ordered categorical. If not given here or in *dtype*, the resulting categorical will be unordered.

**dtype** [CategoricalDtype or “category”, optional] If CategoricalDtype, cannot be used together with *categories* or *ordered*.

**copy** [bool, default False] Make a copy of input ndarray.

**name** [object, optional] Name to be stored in the index.

See also:

**Index** The base Koalas Index type.

## Examples

```
>>> ks.CategoricalIndex(["a", "b", "c", "a", "b", "c"])
CategoricalIndex(['a', 'b', 'c', 'a', 'b', 'c'],
                  categories=['a', 'b', 'c'], ordered=False, dtype='category')
```

CategoricalIndex can also be instantiated from a Categorical:

```
>>> c = pd.Categorical(["a", "b", "c", "a", "b", "c"])
>>> ks.CategoricalIndex(c)
CategoricalIndex(['a', 'b', 'c', 'a', 'b', 'c'],
                  categories=['a', 'b', 'c'], ordered=False, dtype='category')
```

Ordered CategoricalIndex can have a min and max value.

```
>>> ci = ks.CategoricalIndex(
...     ["a", "b", "c", "a", "b", "c"], ordered=True, categories=["c", "b", "a"]
... )
>>> ci
CategoricalIndex(['a', 'b', 'c', 'a', 'b', 'c'],
                  categories=['c', 'b', 'a'], ordered=True, dtype='category')
```

From a Series:

```
>>> s = ks.Series(["a", "b", "c", "a", "b", "c"], index=[10, 20, 30, 40, 50, 60])
>>> ks.CategoricalIndex(s)
CategoricalIndex(['a', 'b', 'c', 'a', 'b', 'c'],
                  categories=['a', 'b', 'c'], ordered=False, dtype='category')
```

From an Index:

```
>>> idx = ks.Index(["a", "b", "c", "a", "b", "c"])
>>> ks.CategoricalIndex(idx)
CategoricalIndex(['a', 'b', 'c', 'a', 'b', 'c'],
                  categories=['a', 'b', 'c'], ordered=False, dtype='category')
```

**\_\_init\_\_** (\*args, \*\*kwargs)

Initialize self. See help(type(self)) for accurate signature.



## Methods

<code>__init__(*args, **kwargs)</code>	Initialize self.
<code>all([axis])</code>	Return whether all elements are True.
<code>any([axis])</code>	Return whether any element is True.
<code>append(other)</code>	Append a collection of Index options together.
<code>argmax()</code>	Return a maximum argument indexer.
<code>argmin()</code>	Return a minimum argument indexer.
<code>asof(label)</code>	Return the label from the index, or, if not present, the previous one.
<code>astype(dtype)</code>	Cast a Koalas object to a specified dtype <code>dtype</code> .
<code>copy([name, deep])</code>	Make a copy of this object.
<code>delete(loc)</code>	Make new Index with passed location(-s) deleted.
<code>difference(other[, sort])</code>	Return a new Index with elements from the index that are not in <i>other</i> .
<code>drop(labels)</code>	Make new Index with passed list of labels deleted.
<code>drop_duplicates()</code>	Return Index with duplicate values removed.
<code>droplevel(level)</code>	Return index with requested level(s) removed.
<code>dropna()</code>	Return Index or MultiIndex without NA/NaN values
<code>equals(other)</code>	Determine if two Index objects contain the same elements.
<code>factorize([sort, na_sentinel])</code>	Encode the object as an enumerated type or categorical variable.
<code>fillna(value)</code>	Fill NA/NaN values with the specified value.
<code>get_level_values(level)</code>	Return Index if a valid level is given.
<code>holds_integer()</code>	Whether the type is an integer type.
<code>identical(other)</code>	Similar to equals, but check that other comparable attributes are also equal.
<code>insert(loc, item)</code>	Make new Index inserting new item at location.
<code>intersection(other)</code>	Form the intersection of two Index objects.
<code>is_boolean()</code>	Return if the current index type is a boolean type.
<code>is_categorical()</code>	Return if the current index type is a categorical type.
<code>is_floating()</code>	Return if the current index type is a floating type.
<code>is_integer()</code>	Return if the current index type is a integer type.
<code>is_interval()</code>	Return if the current index type is an interval type.
<code>is_numeric()</code>	Return if the current index type is a numeric type.
<code>is_object()</code>	Return if the current index type is a object type.
<code>is_type_compatible(kind)</code>	Whether the index type is compatible with the provided type.
<code>isin(values)</code>	Check whether <i>values</i> are contained in Series or Index.
<code>isna()</code>	Detect existing (non-missing) values.
<code>isnull()</code>	Detect existing (non-missing) values.
<code>item()</code>	Return the first element of the underlying data as a python scalar.
<code>max()</code>	Return the maximum value of the Index.
<code>min()</code>	Return the minimum value of the Index.
<code>notna()</code>	Detect existing (non-missing) values.
<code>notnull()</code>	Detect existing (non-missing) values.
<code>nunique([dropna, approx, rsd])</code>	Return number of unique elements in the object.
<code>rename(name[, inplace])</code>	Alter Index or MultiIndex name.

continues on next page

Table 77 – continued from previous page

<code>repeat(repeats)</code>	Repeat elements of a Index/MultiIndex.
<code>set_names(names[, level, inplace])</code>	Set Index or MultiIndex name.
<code>shift([periods, fill_value])</code>	Shift Series/Index by desired number of periods.
<code>sort(*args, **kwargs)</code>	Use <code>sort_values</code> instead.
<code>sort_values([ascending])</code>	Return a sorted copy of the index.
<code>symmetric_difference(other[, result_name, sort])</code>	Compute the symmetric difference of two Index objects.
<code>take(indices)</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>toPandas()</code>	Return a pandas Index.
<code>to_frame([index, name])</code>	Create a DataFrame with a column containing the Index.
<code>to_list()</code>	Return a list of the values.
<code>to_numpy([dtype, copy])</code>	A NumPy ndarray representing the values in this Index or MultiIndex.
<code>to_pandas()</code>	Return a pandas Index.
<code>to_series([name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an index based on an index.
<code>tolist()</code>	Return a list of the values.
<code>transpose()</code>	Return the transpose, For index, It will be index itself.
<code>union(other[, sort])</code>	Form the union of two Index objects.
<code>unique([level])</code>	Return unique values in the index.
<code>value_counts([normalize, sort, ascending, ...])</code>	Return a Series containing counts of unique values.
<code>view()</code>	this is defined as a copy with the same identity

### Attributes

<code>T</code>	Return the transpose, For index, It will be index itself.
<code>asi8</code>	Integer representation of the values.
<code>categories</code>	The categories of this categorical.
<code>codes</code>	The category codes of this categorical.
<code>dtype</code>	Return the dtype object of the underlying data.
<code>empty</code>	Returns true if the current object is empty.
<code>has_duplicates</code>	If index has duplicates, return True, otherwise False.
<code>hasnans</code>	Return True if it has any missing values.
<code>inferred_type</code>	Return a string of the type inferred from the values.
<code>is_all_dates</code>	Return if all data types of the index are datetime.
<code>is_monotonic</code>	Return boolean if values in the object are monotonically increasing.
<code>is_monotonic_decreasing</code>	Return boolean if values in the object are monotonically decreasing.
<code>is_monotonic_increasing</code>	Return boolean if values in the object are monotonically increasing.
<code>is_unique</code>	Return if the index has unique values.
<code>name</code>	Return name of the Index.
<code>names</code>	Return names of the Index.

continues on next page

Table 78 – continued from previous page

<code>ndim</code>	Return an int representing the number of array dimensions.
<code>nlevels</code>	Number of levels in Index & MultiIndex.
<code>ordered</code>	Whether the categories have an ordered relationship.
<code>shape</code>	Return a tuple of the shape of the underlying data.
<code>size</code>	Return an int representing the number of elements in this object.
<code>spark_column</code>	Spark Column object representing the Series/Index.
<code>spark_type</code>	Returns the data type as defined by Spark, as a Spark <code>DataType</code> object.
<code>values</code>	Return an array representing the data in the Index.

## Categorical components

<code>CategoricalIndex.codes</code>	The category codes of this categorical.
<code>CategoricalIndex.categories</code>	The categories of this categorical.
<code>CategoricalIndex.ordered</code>	Whether the categories have an ordered relationship.

## `databricks.koalas.CategoricalIndex.codes`

### property `CategoricalIndex.codes`

The category codes of this categorical.

Codes are an Index of integers which are the positions of the actual values in the categories Index.

There is no setter, use the other categorical methods and the normal item setter to change values in the categorical.

#### Returns

**Index** A non-writable view of the `codes` Index.

## Examples

```
>>> idx = ks.CategoricalIndex(list("abbccc"))
>>> idx
CategoricalIndex(['a', 'b', 'b', 'c', 'c', 'c'],
                  categories=['a', 'b', 'c'], ordered=False, dtype='category')
```

```
>>> idx.codes
Int64Index([0, 1, 1, 2, 2, 2], dtype='int64')
```

**databricks.koalas.CategoricalIndex.categories****property** `CategoricalIndex.categories`

The categories of this categorical.

**Examples**

```
>>> idx = ks.CategoricalIndex(list("abbccc"))
>>> idx
CategoricalIndex(['a', 'b', 'b', 'c', 'c', 'c'],
                  categories=['a', 'b', 'c'], ordered=False, dtype='category')
```

```
>>> idx.categories
Index(['a', 'b', 'c'], dtype='object')
```

**databricks.koalas.CategoricalIndex.ordered****property** `CategoricalIndex.ordered`

Whether the categories have an ordered relationship.

**Examples**

```
>>> idx = ks.CategoricalIndex(list("abbccc"))
>>> idx
CategoricalIndex(['a', 'b', 'b', 'c', 'c', 'c'],
                  categories=['a', 'b', 'c'], ordered=False, dtype='category')
```

```
>>> idx.ordered
False
```

## 3.5.5 MultiIndex

---

*MultiIndex*Koalas MultiIndex that corresponds to pandas MultiIndex logically.

---

**databricks.koalas.MultiIndex****class** `databricks.koalas.MultiIndex`

Koalas MultiIndex that corresponds to pandas MultiIndex logically. This might hold Spark Column internally.

**Parameters****levels** [sequence of arrays] The unique labels for each level.**codes** [sequence of arrays] Integers for each level designating which label at each location.**sortorder** [optional int] Level of sortedness (must be lexicographically sorted by that level).**names** [optional sequence of objects] Names for each of the index levels. (name is accepted for compat).

**copy** [bool, default False] Copy the meta-data.

**verify\_integrity** [bool, default True] Check that the levels/codes are consistent and valid.

See also:

**MultiIndex.from\_arrays** Convert list of arrays to MultiIndex.

**MultiIndex.from\_product** Create a MultiIndex from the cartesian product of iterables.

**MultiIndex.from\_tuples** Convert list of tuples to a MultiIndex.

**MultiIndex.from\_frame** Make a MultiIndex from a DataFrame.

**Index** A single-level Index.

## Examples

```
>>> ks.DataFrame({'a': ['a', 'b', 'c']}, index=[[1, 2, 3], [4, 5, 6]]).index
MultiIndex([(1, 4),
            (2, 5),
            (3, 6)],
           )
```

```
>>> ks.DataFrame({'a': [1, 2, 3]}, index=[list('abc'), list('def')]).index
MultiIndex([('a', 'd'),
            ('b', 'e'),
            ('c', 'f')],
           )
```

**\_\_init\_\_** (\*args, \*\*kwargs)  
Initialize self. See help(type(self)) for accurate signature.

## Methods

<code>__init__</code> (*args, **kwargs)	Initialize self.
<code>all</code> (*args, **kwargs)	Return whether all elements are True.
<code>any</code> (*args, **kwargs)	Return whether any element is True.
<code>append</code> (other)	Append a collection of Index options together.
<code>argmax</code> ()	Return a maximum argument indexer.
<code>argmin</code> ()	Return a minimum argument indexer.
<code>asof</code> (label)	Return the label from the index, or, if not present, the previous one.
<code>astype</code> (dtype)	Cast a Koalas object to a specified dtype dtype.
<code>copy</code> ([deep])	Make a copy of this object.
<code>delete</code> (loc)	Make new Index with passed location(-s) deleted.
<code>difference</code> (other[, sort])	Return a new Index with elements from the index that are not in other.
<code>drop</code> (codes[, level])	Make new MultiIndex with passed list of labels deleted
<code>drop_duplicates</code> ()	Return Index with duplicate values removed.
<code>droplevel</code> (level)	Return index with requested level(s) removed.
<code>dropna</code> ()	Return Index or MultiIndex without NA/NaN values

continues on next page

Table 81 – continued from previous page

<code>equals(other)</code>	Determine if two Index objects contain the same elements.
<code>factorize([sort, na_sentinel])</code>	Encode the object as an enumerated type or categorical variable.
<code>fillna(value)</code>	Fill NA/NaN values with the specified value.
<code>from_arrays(arrays[, sortorder, names])</code>	Convert arrays to MultiIndex.
<code>from_frame(df[, names])</code>	Make a MultiIndex from a DataFrame.
<code>from_product(iterables[, sortorder, names])</code>	Make a MultiIndex from the cartesian product of multiple iterables.
<code>from_tuples(tuples[, sortorder, names])</code>	Convert list of tuples to MultiIndex.
<code>get_level_values(level)</code>	Return vector of label values for requested level, equal to the length of the index.
<code>holds_integer()</code>	Whether the type is an integer type.
<code>identical(other)</code>	Similar to equals, but check that other comparable attributes are also equal.
<code>insert(loc, item)</code>	Make new MultiIndex inserting new item at location.
<code>intersection(other)</code>	Form the intersection of two Index objects.
<code>is_boolean()</code>	Return if the current index type is a boolean type.
<code>is_categorical()</code>	Return if the current index type is a categorical type.
<code>is_floating()</code>	Return if the current index type is a floating type.
<code>is_integer()</code>	Return if the current index type is an integer type.
<code>is_interval()</code>	Return if the current index type is an interval type.
<code>is_numeric()</code>	Return if the current index type is a numeric type.
<code>is_object()</code>	Return if the current index type is a object type.
<code>is_type_compatible(kind)</code>	Whether the index type is compatible with the provided type.
<code>isin(values)</code>	Check whether <i>values</i> are contained in Series or Index.
<code>isna()</code>	Detect existing (non-missing) values.
<code>isnull()</code>	Detect existing (non-missing) values.
<code>item()</code>	Return the first element of the underlying data as a python tuple.
<code>max()</code>	Return the maximum value of the Index.
<code>min()</code>	Return the minimum value of the Index.
<code>notna()</code>	Detect existing (non-missing) values.
<code>notnull()</code>	Detect existing (non-missing) values.
<code>nunique([dropna])</code>	Return number of unique elements in the object.
<code>rename(name[, inplace])</code>	Alter Index or MultiIndex name.
<code>repeat(repeats)</code>	Repeat elements of a Index/MultiIndex.
<code>set_names(names[, level, inplace])</code>	Set Index or MultiIndex name.
<code>shift([periods, fill_value])</code>	Shift Series/Index by desired number of periods.
<code>sort(*args, **kwargs)</code>	Use sort_values instead.
<code>sort_values([ascending])</code>	Return a sorted copy of the index.
<code>swaplevel([i, j])</code>	Swap level i with level j.
<code>symmetric_difference(other[, result_name, sort])</code>	Compute the symmetric difference of two MultiIndex objects.
<code>take(indices)</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>toPandas()</code>	Return a pandas MultiIndex.
<code>to_frame([index, name])</code>	Create a DataFrame with the levels of the MultiIndex as columns.

continues on next page

Table 81 – continued from previous page

<code>to_list()</code>	Return a list of the values.
<code>to_numpy([dtype, copy])</code>	A NumPy ndarray representing the values in this Index or MultiIndex.
<code>to_pandas()</code>	Return a pandas MultiIndex.
<code>to_series([name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.
<code>tolist()</code>	Return a list of the values.
<code>transpose()</code>	Return the transpose, For index, It will be index itself.
<code>union(other[, sort])</code>	Form the union of two Index objects.
<code>unique([level])</code>	Return unique values in the index.
<code>value_counts([normalize, sort, ascending, ...])</code>	Return a Series containing counts of unique values.
<code>view()</code>	this is defined as a copy with the same identity

**Attributes**

<code>T</code>	Return the transpose, For index, It will be index itself.
<code>asi8</code>	Integer representation of the values.
<code>dtype</code>	Return the dtype object of the underlying data.
<code>empty</code>	Returns true if the current object is empty.
<code>has_duplicates</code>	If index has duplicates, return True, otherwise False.
<code>hasnans</code>	Return True if it has any missing values.
<code>inferred_type</code>	Return a string of the type inferred from the values.
<code>is_all_dates</code>	is_all_dates always returns False for MultiIndex
<code>is_monotonic</code>	Return boolean if values in the object are monotonically increasing.
<code>is_monotonic_decreasing</code>	Return boolean if values in the object are monotonically decreasing.
<code>is_monotonic_increasing</code>	Return boolean if values in the object are monotonically increasing.
<code>is_unique</code>	Return if the index has unique values.
<code>levshape</code>	A tuple with the length of each level.
<code>name</code>	Return name of the Index.
<code>names</code>	Return names of the Index.
<code>ndim</code>	Return an int representing the number of array dimensions.
<code>nlevels</code>	Number of levels in Index & MultiIndex.
<code>shape</code>	Return a tuple of the shape of the underlying data.
<code>size</code>	Return an int representing the number of elements in this object.
<code>spark_column</code>	Spark Column object representing the Series/Index.
<code>spark_type</code>	Returns the data type as defined by Spark, as a Spark DataType object.
<code>values</code>	Return an array representing the data in the Index.

## MultiIndex Constructors

<code>MultiIndex.from_arrays</code> (arrays[, ...])	sortorder,	Convert arrays to MultiIndex.
<code>MultiIndex.from_tuples</code> (tuples[, ...])	sortorder,	Convert list of tuples to MultiIndex.
<code>MultiIndex.from_product</code> (iterables[, ...])		Make a MultiIndex from the cartesian product of multiple iterables.
<code>MultiIndex.from_frame</code> (df[, names])		Make a MultiIndex from a DataFrame.

### `databricks.koalas.MultiIndex.from_arrays`

**static** `MultiIndex.from_arrays` (arrays, sortorder=None, names=None) →  
 databricks.koalas.indexes.multi.MultiIndex  
 Convert arrays to MultiIndex.

#### Parameters

**arrays:** list / sequence of array-likes Each array-like gives one level's value for each data point. `len(arrays)` is the number of levels.

**sortorder:** int or None Level of sortedness (must be lexicographically sorted by that level).

**names:** list / sequence of str, optional Names for the levels in the index.

#### Returns

**index:** MultiIndex

#### Examples

```
>>> arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
>>> ks.MultiIndex.from_arrays(arrays, names=('number', 'color'))
MultiIndex([(1, 'red'),
            (1, 'blue'),
            (2, 'red'),
            (2, 'blue')],
            names=['number', 'color'])
```

### `databricks.koalas.MultiIndex.from_tuples`

**static** `MultiIndex.from_tuples` (tuples, sortorder=None, names=None) →  
 databricks.koalas.indexes.multi.MultiIndex  
 Convert list of tuples to MultiIndex.

#### Parameters

**tuples** [list / sequence of tuple-likes] Each tuple is the index of one row/column.

**sortorder** [int or None] Level of sortedness (must be lexicographically sorted by that level).

**names** [list / sequence of str, optional] Names for the levels in the index.

#### Returns

**index** [MultiIndex]



## Examples

```
>>> tuples = [(1, 'red'), (1, 'blue'),
...           (2, 'red'), (2, 'blue')]
>>> ks.MultiIndex.from_tuples(tuples, names=('number', 'color'))
MultiIndex([(1, 'red'),
            (1, 'blue'),
            (2, 'red'),
            (2, 'blue')],
           names=['number', 'color'])
```

## databricks.koalas.MultiIndex.from\_product

**static** `MultiIndex.from_product(iterables, sortorder=None, names=None)` → `databricks.koalas.indexes.multi.MultiIndex`  
 Make a MultiIndex from the cartesian product of multiple iterables.

### Parameters

- iterables** [list / sequence of iterables] Each iterable has unique labels for each level of the index.
- sortorder** [int or None] Level of sortedness (must be lexicographically sorted by that level).
- names** [list / sequence of str, optional] Names for the levels in the index.

### Returns

**index** [MultiIndex]

See also:

**`MultiIndex.from_arrays`** Convert list of arrays to MultiIndex.

**`MultiIndex.from_tuples`** Convert list of tuples to MultiIndex.

## Examples

```
>>> numbers = [0, 1, 2]
>>> colors = ['green', 'purple']
>>> ks.MultiIndex.from_product([numbers, colors],
...                             names=['number', 'color'])
MultiIndex([(0, 'green'),
            (0, 'purple'),
            (1, 'green'),
            (1, 'purple'),
            (2, 'green'),
            (2, 'purple')],
           names=['number', 'color'])
```

**databricks.koalas.MultiIndex.from\_frame**

**static** `MultiIndex.from_frame(df, names=None)` → `databricks.koalas.indexes.multi.MultiIndex`  
 Make a MultiIndex from a DataFrame.

**Parameters**

**df** [DataFrame] DataFrame to be converted to MultiIndex.

**names** [list-like, optional] If no names are provided, use the column names, or tuple of column names if the columns is a MultiIndex. If a sequence, overwrite names with the given sequence.

**Returns**

**MultiIndex** The MultiIndex representation of the given DataFrame.

See also:

***MultiIndex.from\_arrays*** Convert list of arrays to MultiIndex.

***MultiIndex.from\_tuples*** Convert list of tuples to MultiIndex.

***MultiIndex.from\_product*** Make a MultiIndex from cartesian product of iterables.

**Examples**

```
>>> df = ks.DataFrame([['HI', 'Temp'], ['HI', 'Precip'],
...                    ['NJ', 'Temp'], ['NJ', 'Precip']],
...                    columns=['a', 'b'])
>>> df
   a      b
0  HI  Temp
1  HI Precip
2  NJ  Temp
3  NJ Precip
```

```
>>> ks.MultiIndex.from_frame(df)
MultiIndex([('HI', 'Temp'),
            ('HI', 'Precip'),
            ('NJ', 'Temp'),
            ('NJ', 'Precip')],
            names=['a', 'b'])
```

Using explicit names, instead of the column names

```
>>> ks.MultiIndex.from_frame(df, names=['state', 'observation'])
MultiIndex([('HI', 'Temp'),
            ('HI', 'Precip'),
            ('NJ', 'Temp'),
            ('NJ', 'Precip')],
            names=['state', 'observation'])
```

## MultIndex Properties

<code>MultiIndex.has_duplicates</code>	If index has duplicates, return True, otherwise False.
<code>MultiIndex.hasnans</code>	Return True if it has any missing values.
<code>MultiIndex.inferred_type</code>	Return a string of the type inferred from the values.
<code>MultiIndex.is_all_dates</code>	<code>is_all_dates</code> always returns False for MultiIndex
<code>MultiIndex.shape</code>	Return a tuple of the shape of the underlying data.
<code>MultiIndex.names</code>	Return names of the Index.
<code>MultiIndex.ndim</code>	Return an int representing the number of array dimensions.
<code>MultiIndex.empty</code>	Returns true if the current object is empty.
<code>MultiIndex.T</code>	Return the transpose, For index, It will be index itself.
<code>MultiIndex.size</code>	Return an int representing the number of elements in this object.
<code>MultiIndex.nlevels</code>	Number of levels in Index & MultiIndex.
<code>MultiIndex.levshape</code>	A tuple with the length of each level.
<code>MultiIndex.values</code>	Return an array representing the data in the Index.

### `databricks.koalas.MultiIndex.has_duplicates`

**property** `MultiIndex.has_duplicates`

If index has duplicates, return True, otherwise False.

### Examples

```
>>> idx = ks.Index([1, 5, 7, 7])
>>> idx.has_duplicates
True
```

```
>>> idx = ks.Index([1, 5, 7])
>>> idx.has_duplicates
False
```

```
>>> idx = ks.Index(["Watermelon", "Orange", "Apple",
...                 "Watermelon"])
>>> idx.has_duplicates
True
```

```
>>> idx = ks.Index(["Orange", "Apple",
...                 "Watermelon"])
>>> idx.has_duplicates
False
```

**databricks.koalas.MultiIndex.hasnans****property** MultiIndex.**hasnans**

Return True if it has any missing values. Otherwise, it returns False.

```
>>> ks.DataFrame({}, index=list('abc')).index.hasnans
False
```

```
>>> ks.Series(['a', None]).hasnans
True
```

```
>>> ks.Series([1.0, 2.0, np.nan]).hasnans
True
```

```
>>> ks.Series([1, 2, 3]).hasnans
False
```

```
>>> (ks.Series([1.0, 2.0, np.nan]) + 1).hasnans
True
```

```
>>> ks.Series([1, 2, 3]).rename("a").to_frame().set_index("a").index.hasnans
False
```

**databricks.koalas.MultiIndex.inferred\_type****property** MultiIndex.**inferred\_type**

Return a string of the type inferred from the values.

**databricks.koalas.MultiIndex.is\_all\_dates****property** MultiIndex.**is\_all\_dates**

is\_all\_dates always returns False for MultiIndex

**Examples**

```
>>> from datetime import datetime
```

```
>>> idx = ks.MultiIndex.from_tuples(
...     [(datetime(2019, 1, 1, 0, 0, 0), datetime(2019, 1, 1, 0, 0, 0)),
...      (datetime(2019, 1, 1, 0, 0, 0), datetime(2019, 1, 1, 0, 0, 0))]
... )
>>> idx
MultiIndex([('2019-01-01', '2019-01-01'),
            ('2019-01-01', '2019-01-01')],
           )
```

```
>>> idx.is_all_dates
False
```

**databricks.koalas.MultiIndex.shape****property** MultiIndex.**shape**

Return a tuple of the shape of the underlying data.

**Examples**

```
>>> idx = ks.Index(['a', 'b', 'c'])
>>> idx
Index(['a', 'b', 'c'], dtype='object')
>>> idx.shape
(3,)
```

```
>>> midx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> midx
MultiIndex([('a', 'x'),
            ('b', 'y'),
            ('c', 'z')],
           )
>>> midx.shape
(3,)
```

**databricks.koalas.MultiIndex.names****property** MultiIndex.**names**

Return names of the Index.

**databricks.koalas.MultiIndex.ndim****property** MultiIndex.**ndim**

Return an int representing the number of array dimensions.

Return 1 for Series / Index / MultiIndex.

**Examples**

For Series

```
>>> s = ks.Series([None, 1, 2, 3, 4], index=[4, 5, 2, 1, 8])
>>> s.ndim
1
```

For Index

```
>>> s.index.ndim
1
```

For MultiIndex

```
>>> midx = pd.MultiIndex([['lama', 'cow', 'falcon'],
...                       ['speed', 'weight', 'length']],
...                      [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                      [1, 1, 1, 1, 1, 2, 1, 2, 2]])
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3], index=midx)
>>> s.index.ndim
1
```

### **databricks.koalas.MultiIndex.empty**

#### **property** MultiIndex.empty

Returns true if the current object is empty. Otherwise, returns false.

```
>>> ks.range(10).id.empty
False
```

```
>>> ks.range(0).id.empty
True
```

```
>>> ks.DataFrame({}, index=list('abc')).index.empty
False
```

### **databricks.koalas.MultiIndex.T**

#### **property** MultiIndex.T

Return the transpose, For index, It will be index itself.

#### **Examples**

```
>>> idx = ks.Index(['a', 'b', 'c'])
>>> idx
Index(['a', 'b', 'c'], dtype='object')
```

```
>>> idx.transpose()
Index(['a', 'b', 'c'], dtype='object')
```

#### **For MultiIndex**

```
>>> midx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> midx
MultiIndex([('a', 'x'),
            ('b', 'y'),
            ('c', 'z')],
           )
```

```
>>> midx.transpose()
MultiIndex([('a', 'x'),
            ('b', 'y'),
            ('c', 'z')],
           )
```

**databricks.koalas.MultiIndex.size****property** MultiIndex.size

Return an int representing the number of elements in this object.

**Examples**

```
>>> df = ks.DataFrame([(.2, .3), (.0, .6), (.6, .0), (.2, .1)],
...                     columns=['dogs', 'cats'],
...                     index=list('abcd'))
>>> df.index.size
4
```

```
>>> df.set_index('dogs', append=True).index.size
4
```

**databricks.koalas.MultiIndex.nlevels****property** MultiIndex.nlevels

Number of levels in Index & MultiIndex.

**Examples**

```
>>> kdf = ks.DataFrame({"a": [1, 2, 3]}, index=pd.Index(['a', 'b', 'c'], name="idx
↪"))
>>> kdf.index.nlevels
1
```

```
>>> kdf = ks.DataFrame({'a': [1, 2, 3]}, index=[list('abc'), list('def')])
>>> kdf.index.nlevels
2
```

**databricks.koalas.MultiIndex.levshape****property** MultiIndex.levshape

A tuple with the length of each level.

**Examples**

```
>>> midx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> midx
MultiIndex([('a', 'x'),
            ('b', 'y'),
            ('c', 'z')],
           )
```

```
>>> midx.levshape
(3, 3)
```

**databricks.koalas.MultiIndex.values****property** `MultiIndex.values`

Return an array representing the data in the Index.

**Warning:** We recommend using `Index.to_numpy()` instead.

---

**Note:** This method should only be used if the resulting NumPy ndarray is expected to be small, as all the data is loaded into the driver's memory.

---

**Returns****numpy.ndarray****Examples**

```
>>> ks.Series([1, 2, 3, 4]).index.values
array([0, 1, 2, 3])
>>> ks.DataFrame({'a': ['a', 'b', 'c']}, index=[[1, 2, 3], [4, 5, 6]]).index.
↪values
array([(1, 4), (2, 5), (3, 6)], dtype=object)
```

**MultiIndex components**

---

`MultiIndex.swaplevel(i, j)`Swap level *i* with level *j*.

---

**databricks.koalas.MultiIndex.swaplevel**`MultiIndex.swaplevel` (*i*=-2, *j*=-1) → `databricks.koalas.indexes.multi.MultiIndex`Swap level *i* with level *j*. Calling this method does not change the ordering of the values.**Parameters****i** [int, str, default -2] First level of index to be swapped. Can pass level name as string. Type of parameters can be mixed.**j** [int, str, default -1] Second level of index to be swapped. Can pass level name as string. Type of parameters can be mixed.**Returns****MultiIndex** A new MultiIndex.



## Examples

```
>>> midx = ks.MultiIndex.from_arrays([[ 'a', 'b'], [1, 2]], names = [ 'word',
↳ 'number'])
>>> midx
MultiIndex([('a', 1),
            ('b', 2)],
            names=[ 'word', 'number'])
```

```
>>> midx.swaplevel(0, 1)
MultiIndex([(1, 'a'),
            (2, 'b')],
            names=[ 'number', 'word'])
```

```
>>> midx.swaplevel('number', 'word')
MultiIndex([(1, 'a'),
            (2, 'b')],
            names=[ 'number', 'word'])
```

## MultiIndex components

<code>MultiIndex.droplevel(level)</code>	Return index with requested level(s) removed.
--	---

## databricks.koalas.MultiIndex.droplevel

`MultiIndex.droplevel(level)` → `databricks.koalas.indexes.base.Index`

Return index with requested level(s) removed. If resulting index has only 1 level left, the result will be of Index type, not MultiIndex.

### Parameters

**level** [int, str, tuple, or list-like, default 0] If a string is given, must be the name of a level If list-like, elements must be names or indexes of levels.

### Returns

**Index or MultiIndex**

## Examples

```
>>> midx = ks.DataFrame({'a': [ 'a', 'b']}, index=[[ 'a', 'x'], [ 'b', 'y'], [1, 2]]).index
↳
>>> midx
MultiIndex([('a', 'b', 1),
            ('x', 'y', 2)],
            )
>>> midx.droplevel([0, 1])
Int64Index([1, 2], dtype='int64')
>>> midx.droplevel(0)
MultiIndex([('b', 1),
            ('y', 2)],
            )
```

(continues on next page)

(continued from previous page)

```
>>> midx.names = [("a", "b"), "b", "c"]
>>> midx.droplevel(['a', 'b'])
MultiIndex([('b', 1),
            ('y', 2)],
            names=['b', 'c'])
```

## MultiIndex Missing Values

<code>MultiIndex.fillna(value)</code>	Fill NA/NaN values with the specified value.
<code>MultiIndex.dropna()</code>	Return Index or MultiIndex without NA/NaN values

### databricks.koalas.MultiIndex.fillna

`MultiIndex.fillna(value: Union[int, float, str, bytes, decimal.Decimal, datetime.date, None])` → `databricks.koalas.indexes.base.Index`  
 Fill NA/NaN values with the specified value.

#### Parameters

**value** [scalar] Scalar value to use to fill holes (e.g. 0). This value cannot be a list-likes.

#### Returns

**Index** : filled with value

#### Examples

```
>>> ki = ks.DataFrame({'a': ['a', 'b', 'c']}, index=[1, 2, None]).index
>>> ki
Float64Index([1.0, 2.0, nan], dtype='float64')
```

```
>>> ki.fillna(0)
Float64Index([1.0, 2.0, 0.0], dtype='float64')
```

### databricks.koalas.MultiIndex.dropna

`MultiIndex.dropna()` → `databricks.koalas.indexes.base.Index`  
 Return Index or MultiIndex without NA/NaN values

#### Examples

```
>>> df = ks.DataFrame([[1, 2], [4, 5], [7, 8]],
...                   index=['cobra', 'viper', None],
...                   columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
cobra         1      2
viper         4      5
NaN           7      8
```

```
>>> df.index.dropna()
Index(['cobra', 'viper'], dtype='object')
```

Also support for MultiIndex

```
>>> midx = pd.MultiIndex([[ 'lama', 'cow', 'falcon'],
...                       [None, 'weight', 'length']],
...                       [[0, 1, 1, 1, 1, 1, 2, 2, 2],
...                       [0, 1, 1, 0, 1, 2, 1, 1, 2]])
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, None],
...               index=midx)
>>> s
lama      NaN      45.0
cow      weight  200.0
         weight   1.2
         NaN     30.0
         weight  250.0
         length   1.5
falcon  weight  320.0
         weight   1.0
         length   NaN
dtype: float64
```

```
>>> s.index.dropna()
MultiIndex([( 'cow', 'weight'),
            ( 'cow', 'weight'),
            ( 'cow', 'weight'),
            ( 'cow', 'length'),
            ('falcon', 'weight'),
            ('falcon', 'weight'),
            ('falcon', 'length')],
          )
```

## MultIndex Modifying and computations

<code>MultiIndex.equals(other)</code>	Determine if two Index objects contain the same elements.
<code>MultiIndex.identical(other)</code>	Similar to equals, but check that other comparable attributes are also equal.
<code>MultiIndex.insert(loc, item)</code>	Make new MultiIndex inserting new item at location.
<code>MultiIndex.drop(codes[, level])</code>	Make new MultiIndex with passed list of labels deleted
<code>MultiIndex.copy([deep])</code>	Make a copy of this object.
<code>MultiIndex.delete(loc)</code>	Make new Index with passed location(-s) deleted.
<code>MultiIndex.rename(name[, inplace])</code>	Alter Index or MultiIndex name.
<code>MultiIndex.repeat(repeats)</code>	Repeat elements of a Index/MultiIndex.
<code>MultiIndex.take(indices)</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>MultiIndex.unique([level])</code>	Return unique values in the index.
<code>MultiIndex.min()</code>	Return the minimum value of the Index.
<code>MultiIndex.max()</code>	Return the maximum value of the Index.
<code>MultiIndex.value_counts([normalize, sort, ...])</code>	Return a Series containing counts of unique values.

**databricks.koalas.MultiIndex.equals**

`MultiIndex.equals` (*other*) → bool

Determine if two Index objects contain the same elements.

**Returns**

**bool** True if “other” is an Index and it has the same elements as calling index; False otherwise.

**Examples**

```
>>> from databricks.koalas.config import option_context
>>> idx = ks.Index(['a', 'b', 'c'])
>>> idx.name = "name"
>>> midx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> midx.names = ("nameA", "nameB")
```

**For Index**

```
>>> idx.equals(idx)
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     idx.equals(ks.Index(['a', 'b', 'c']))
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     idx.equals(ks.Index(['b', 'b', 'a']))
False
>>> idx.equals(midx)
False
```

**For MultiIndex**

```
>>> midx.equals(midx)
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     midx.equals(ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')
...     ↪ 'x')]))
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     midx.equals(ks.MultiIndex.from_tuples([('c', 'z'), ('b', 'y'), ('a', 'x')
...     ↪ 'x')]))
False
>>> midx.equals(idx)
False
```

**databricks.koalas.MultiIndex.identical**

`MultiIndex.identical` (*other*) → bool

Similar to equals, but check that other comparable attributes are also equal.

**Returns**

**bool** If two Index objects have equal elements and same type True, otherwise False.

## Examples

```
>>> from databricks.koalas.config import option_context
>>> idx = ks.Index(['a', 'b', 'c'])
>>> midx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
```

### For Index

```
>>> idx.identical(idx)
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     idx.identical(ks.Index(['a', 'b', 'c']))
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     idx.identical(ks.Index(['b', 'b', 'a']))
False
>>> idx.identical(midx)
False
```

### For MultiIndex

```
>>> midx.identical(midx)
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     midx.identical(ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z
↪')]))
True
>>> with option_context('compute.ops_on_diff_frames', True):
...     midx.identical(ks.MultiIndex.from_tuples([('c', 'z'), ('b', 'y'), ('a', 'x
↪')]))
False
>>> midx.identical(idx)
False
```

## databricks.koalas.MultiIndex.insert

`MultiIndex.insert(loc: int, item) → databricks.koalas.indexes.base.Index`  
 Make new MultiIndex inserting new item at location.

Follows Python list.append semantics for negative values.

### Parameters

**loc** [int]

**item** [object]

### Returns

**new\_index** [MultiIndex]

## Examples

```
>>> kmidx = ks.MultiIndex.from_tuples([("a", "x"), ("b", "y"), ("c", "z")])
>>> kmidx.insert(3, ("h", "j"))
MultiIndex([('a', 'x'),
            ('b', 'y'),
            ('c', 'z'),
            ('h', 'j')],
           )
```

For negative values

```
>>> kmidx.insert(-2, ("h", "j"))
MultiIndex([('a', 'x'),
            ('h', 'j'),
            ('b', 'y'),
            ('c', 'z')],
           )
```

## databricks.koalas.MultiIndex.drop

`MultiIndex.drop(codes, level=None) → databricks.koalas.indexes.multi.MultiIndex`

Make new MultiIndex with passed list of labels deleted

### Parameters

**codes** [array-like] Must be a list of tuples

**level** [int or level name, default None]

### Returns

**dropped** [MultiIndex]

## Examples

```
>>> index = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y'), ('c', 'z')])
>>> index
MultiIndex([('a', 'x'),
            ('b', 'y'),
            ('c', 'z')],
           )
```

```
>>> index.drop(['a'])
MultiIndex([('b', 'y'),
            ('c', 'z')],
           )
```

```
>>> index.drop(['x', 'y'], level=1)
MultiIndex([('c', 'z')],
           )
```

**databricks.koalas.MultiIndex.copy**

`MultiIndex.copy` (*deep=None*) → `databricks.koalas.indexes.multi.MultiIndex`  
 Make a copy of this object.

**Parameters**

**deep** [None] this parameter is not supported but just dummy parameter to match pandas.

**Examples**

```
>>> df = ks.DataFrame([(.2, .3), (.0, .6), (.6, .0), (.2, .1)],
...                    columns=['dogs', 'cats'],
...                    index=[list('abcd'), list('efgh')])
>>> df['dogs'].index
MultiIndex([('a', 'e'),
            ('b', 'f'),
            ('c', 'g'),
            ('d', 'h')],
           )
```

**Copy index**

```
>>> df.index.copy()
MultiIndex([('a', 'e'),
            ('b', 'f'),
            ('c', 'g'),
            ('d', 'h')],
           )
```

**databricks.koalas.MultiIndex.delete**

`MultiIndex.delete` (*loc*) → `databricks.koalas.indexes.base.Index`  
 Make new Index with passed location(-s) deleted.

---

**Note:** this API can be pretty expensive since it is based on a global sequence internally.

---

**Returns**

**new\_index** [Index]

**Examples**

```
>>> kidx = ks.Index([10, 10, 9, 8, 4, 2, 4, 4, 2, 2, 10, 10])
>>> kidx
Int64Index([10, 10, 9, 8, 4, 2, 4, 4, 2, 2, 10, 10], dtype='int64')
```

```
>>> kidx.delete(0).sort_values()
Int64Index([2, 2, 2, 4, 4, 4, 8, 9, 10, 10, 10], dtype='int64')
```

```
>>> kidx.delete([0, 1, 2, 3, 10, 11]).sort_values()
Int64Index([2, 2, 2, 4, 4, 4], dtype='int64')
```

### MultiIndex

```
>>> kidx = ks.MultiIndex.from_tuples([('a', 'x', 1), ('b', 'y', 2), ('c', 'z', 3)])
>>> kidx
MultiIndex([('a', 'x', 1),
            ('b', 'y', 2),
            ('c', 'z', 3)],
           )
```

```
>>> kidx.delete([0, 2]).sort_values()
MultiIndex([('b', 'y', 2)],
           )
```

## databricks.koalas.MultiIndex.rename

`MultiIndex.rename` (*name*: Union[Any, Tuple, List[Union[Any, Tuple]]], *inplace*: bool = False) → Optional[databricks.koalas.indexes.base.Index]

Alter Index or MultiIndex name. Able to set new names without level. Defaults to returning new index.

### Parameters

**name** [label or list of labels] Name(s) to set.

**inplace** [boolean, default False] Modifies the object directly, instead of creating a new Index or MultiIndex.

### Returns

**Index or MultiIndex** The same type as the caller or None if inplace is True.

## Examples

```
>>> df = ks.DataFrame({'a': ['A', 'C'], 'b': ['A', 'B']}, columns=['a', 'b'])
>>> df.index.rename("c")
Int64Index([0, 1], dtype='int64', name='c')
```

```
>>> df.set_index("a", inplace=True)
>>> df.index.rename("d")
Index(['A', 'C'], dtype='object', name='d')
```

You can also change the index name in place.

```
>>> df.index.rename("e", inplace=True)
>>> df.index
Index(['A', 'C'], dtype='object', name='e')
```

```
>>> df
   b
e
A  A
C  B
```



## Support for MultiIndex

```
>>> kidx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y')])
>>> kidx.names = ['hello', 'koalas']
>>> kidx
MultiIndex([('a', 'x'),
            ('b', 'y')],
            names=['hello', 'koalas'])
```

```
>>> kidx.rename(['aloha', 'databricks'])
MultiIndex([('a', 'x'),
            ('b', 'y')],
            names=['aloha', 'databricks'])
```

**databricks.koalas.MultiIndex.repeat**

`MultiIndex.repeat(repeats: int) → databricks.koalas.indexes.base.Index`

Repeat elements of a Index/MultiIndex.

Returns a new Index/MultiIndex where each element of the current Index/MultiIndex is repeated consecutively a given number of times.

**Parameters**

**repeats** [int] The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty Index.

**Returns**

**repeated\_index** [Index/MultiIndex] Newly created Index/MultiIndex with repeated elements.

**See also:**

[`Series.repeat`](#) Equivalent function for Series.

**Examples**

```
>>> idx = ks.Index(['a', 'b', 'c'])
>>> idx
Index(['a', 'b', 'c'], dtype='object')
>>> idx.repeat(2)
Index(['a', 'b', 'c', 'a', 'b', 'c'], dtype='object')
```

For MultiIndex,

```
>>> midx = ks.MultiIndex.from_tuples([('x', 'a'), ('x', 'b'), ('y', 'c')])
>>> midx
MultiIndex([('x', 'a'),
            ('x', 'b'),
            ('y', 'c')],
            )
>>> midx.repeat(2)
MultiIndex([('x', 'a'),
            ('x', 'b'),
            ('y', 'c'),
            ('x', 'a'),
            ('x', 'b'),
            ('y', 'c')],
            )
```

(continues on next page)

(continued from previous page)

```

        ('x', 'b'),
        ('y', 'c')],
    )
>>> midx.repeat(0)
MultiIndex([], )

```

### `databricks.koalas.MultiIndex.take`

`MultiIndex.take(indices)` → Union[databricks.koalas.series.Series, databricks.koalas.indexes.base.Index]

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

#### Parameters

**indices** [array-like] An array of ints indicating which positions to take.

#### Returns

**taken** [same type as caller] An array-like containing the elements taken from the object.

See also:

[`DataFrame.loc`](#) Select a subset of a DataFrame by labels.

[`DataFrame.iloc`](#) Select a subset of a DataFrame by positions.

[`numpy.take`](#) Take elements from an array along an axis.

### Examples

Series

```

>>> kser = ks.Series([100, 200, 300, 400, 500])
>>> kser
0    100
1    200
2    300
3    400
4    500
dtype: int64

```

```

>>> kser.take([0, 2, 4]).sort_index()
0    100
2    300
4    500
dtype: int64

```

Index

```

>>> kidx = ks.Index([100, 200, 300, 400, 500])
>>> kidx
Int64Index([100, 200, 300, 400, 500], dtype='int64')

```

```
>>> kidx.take([0, 2, 4]).sort_values()
Int64Index([100, 300, 500], dtype='int64')
```

### MultiIndex

```
>>> kmidx = ks.MultiIndex.from_tuples([("x", "a"), ("x", "b"), ("x", "c")])
>>> kmidx
MultiIndex([('x', 'a'),
            ('x', 'b'),
            ('x', 'c')],
           )
```

```
>>> kmidx.take([0, 2])
MultiIndex([('x', 'a'),
            ('x', 'c')],
           )
```

## databricks.koalas.MultiIndex.unique

`MultiIndex.unique(level=None)` → `databricks.koalas.indexes.base.Index`

Return unique values in the index.

Be aware the order of unique values might be different than `pandas.Index.unique`

### Parameters

**level** [int or str, optional, default is None]

### Returns

**Index without duplicates**

See also:

[\*Series.unique\*](#)

[\*groupby.SeriesGroupBy.unique\*](#)

### Examples

```
>>> ks.DataFrame({'a': ['a', 'b', 'c']}, index=[1, 1, 3]).index.unique().sort_
↪values()
Int64Index([1, 3], dtype='int64')
```

```
>>> ks.DataFrame({'a': ['a', 'b', 'c']}, index=['d', 'e', 'e']).index.unique().
↪sort_values()
Index(['d', 'e'], dtype='object')
```

### MultiIndex

```
>>> ks.MultiIndex.from_tuples([("A", "X"), ("A", "Y"), ("A", "X")]).unique()
...
MultiIndex([('A', 'X'),
            ('A', 'Y')],
           )
```

**databricks.koalas.MultiIndex.min**

`MultiIndex.min()` → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Tuple[Union[int, float, str, bytes, decimal.Decimal, datetime.date, None], ...]]  
Return the minimum value of the Index.

**Returns**

**scalar** Minimum value.

**See also:**

**`Index.max`** Return the maximum value of the object.

**`Series.min`** Return the minimum value in a Series.

**`DataFrame.min`** Return the minimum values in a DataFrame.

**Examples**

```
>>> idx = ks.Index([3, 2, 1])
>>> idx.min()
1
```

```
>>> idx = ks.Index(['c', 'b', 'a'])
>>> idx.min()
'a'
```

For a MultiIndex, the maximum is determined lexicographically.

```
>>> idx = ks.MultiIndex.from_tuples([('a', 'x', 1), ('b', 'y', 2)])
>>> idx.min()
('a', 'x', 1)
```

**databricks.koalas.MultiIndex.max**

`MultiIndex.max()` → Union[int, float, str, bytes, decimal.Decimal, datetime.date, None, Tuple[Union[int, float, str, bytes, decimal.Decimal, datetime.date, None], ...]]  
Return the maximum value of the Index.

**Returns**

**scalar** Maximum value.

**See also:**

**`Index.min`** Return the minimum value in an Index.

**`Series.max`** Return the maximum value in a Series.

**`DataFrame.max`** Return the maximum values in a DataFrame.

## Examples

```
>>> idx = ks.Index([3, 2, 1])
>>> idx.max()
3
```

```
>>> idx = ks.Index(['c', 'b', 'a'])
>>> idx.max()
'c'
```

For a MultiIndex, the maximum is determined lexicographically.

```
>>> idx = ks.MultiIndex.from_tuples([('a', 'x', 1), ('b', 'y', 2)])
>>> idx.max()
('b', 'y', 2)
```

## databricks.koalas.MultiIndex.value\_counts

`MultiIndex.value_counts` (*normalize=False, sort=True, ascending=False, bins=None, dropna=True*)  
 → `databricks.koalas.series.Series`

Return a Series containing counts of unique values. The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

### Parameters

- normalize** [boolean, default False] If True then the object returned will contain the relative frequencies of the unique values.
- sort** [boolean, default True] Sort by values.
- ascending** [boolean, default False] Sort in ascending order.
- bins** [Not Yet Supported]
- dropna** [boolean, default True] Don't include counts of NaN.

### Returns

**counts** [Series]

See also:

[`Series.count`](#) Number of non-NA elements in a Series.

## Examples

For Series

```
>>> df = ks.DataFrame({'x': [0, 0, 1, 1, 1, np.nan]})
>>> df.x.value_counts()
1.0    3
0.0    2
Name: x, dtype: int64
```

With *normalize* set to *True*, returns the relative frequency by dividing all values by the sum of values.

```
>>> df.x.value_counts(normalize=True)
1.0    0.6
0.0    0.4
Name: x, dtype: float64
```

**dropna** With *dropna* set to *False* we can also see NaN index values.

```
>>> df.x.value_counts(dropna=False)
1.0    3
0.0    2
NaN    1
Name: x, dtype: int64
```

**For Index**

```
>>> idx = ks.Index([3, 1, 2, 3, 4, np.nan])
>>> idx
Float64Index([3.0, 1.0, 2.0, 3.0, 4.0, nan], dtype='float64')
```

```
>>> idx.value_counts().sort_index()
1.0    1
2.0    1
3.0    2
4.0    1
dtype: int64
```

**sort**

With *sort* set to *False*, the result wouldn't be sorted by number of count.

```
>>> idx.value_counts(sort=True).sort_index()
1.0    1
2.0    1
3.0    2
4.0    1
dtype: int64
```

**normalize**

With *normalize* set to *True*, returns the relative frequency by dividing all values by the sum of values.

```
>>> idx.value_counts(normalize=True).sort_index()
1.0    0.2
2.0    0.2
3.0    0.4
4.0    0.2
dtype: float64
```

**dropna**

With *dropna* set to *False* we can also see NaN index values.

```
>>> idx.value_counts(dropna=False).sort_index()
1.0    1
2.0    1
3.0    2
4.0    1
```

(continues on next page)

(continued from previous page)

```
NaN      1
dtype: int64
```

For MultiIndex.

```
>>> midx = pd.MultiIndex([['lama', 'cow', 'falcon'],
...                       ['speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [1, 1, 1, 1, 1, 2, 1, 2, 2]])
>>> s = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3], index=midx)
>>> s.index
MultiIndex([( 'lama', 'weight'),
             ( 'lama', 'weight'),
             ( 'lama', 'weight'),
             ( 'cow', 'weight'),
             ( 'cow', 'weight'),
             ( 'cow', 'length'),
             ('falcon', 'weight'),
             ('falcon', 'length'),
             ('falcon', 'length')],
           )
```

```
>>> s.index.value_counts().sort_index()
(cow, length)      1
(cow, weight)      2
(falcon, length)   2
(falcon, weight)   1
(lama, weight)     3
dtype: int64
```

```
>>> s.index.value_counts(normalize=True).sort_index()
(cow, length)      0.111111
(cow, weight)      0.222222
(falcon, length)   0.222222
(falcon, weight)   0.111111
(lama, weight)     0.333333
dtype: float64
```

If Index has name, keep the name up.

```
>>> idx = ks.Index([0, 0, 0, 1, 1, 2, 3], name='koalas')
>>> idx.value_counts().sort_index()
0      3
1      2
2      1
3      1
Name: koalas, dtype: int64
```

## MultIndex Combining / joining / set operations

<code>MultiIndex.append(other)</code>	Append a collection of Index options together.
<code>MultiIndex.intersection(other)</code>	Form the intersection of two Index objects.
<code>MultiIndex.union(other[, sort])</code>	Form the union of two Index objects.
<code>MultiIndex.difference(other[, sort])</code>	Return a new Index with elements from the index that are not in <i>other</i> .
<code>MultiIndex.symmetric_difference(other[, ...])</code>	Compute the symmetric difference of two MultiIndex objects.

### databricks.koalas.MultiIndex.append

`MultiIndex.append` (*other*: `databricks.koalas.indexes.base.Index`) →  
`databricks.koalas.indexes.base.Index`  
 Append a collection of Index options together.

#### Parameters

**other** [Index]

#### Returns

**appended** [Index]

### Examples

```
>>> kidx = ks.Index([10, 5, 0, 5, 10, 5, 0, 10])
>>> kidx
Int64Index([10, 5, 0, 5, 10, 5, 0, 10], dtype='int64')
```

```
>>> kidx.append(kidx)
Int64Index([10, 5, 0, 5, 10, 5, 0, 10, 10, 5, 0, 5, 10, 5, 0, 10], dtype='int64')
```

### Support for MiltiIndex

```
>>> kidx = ks.MultiIndex.from_tuples([('a', 'x'), ('b', 'y')])
>>> kidx
MultiIndex([('a', 'x'),
             ('b', 'y')],
           )
```

```
>>> kidx.append(kidx)
MultiIndex([('a', 'x'),
             ('b', 'y'),
             ('a', 'x'),
             ('b', 'y')],
           )
```



**databricks.koalas.MultiIndex.intersection**

`MultiIndex.intersection` (*other*) → `databricks.koalas.indexes.multi.MultiIndex`

Form the intersection of two Index objects.

This returns a new Index with elements common to the index and *other*.

**Parameters**

**other** [Index or array-like]

**Returns**

**intersection** [MultiIndex]

**Examples**

```
>>> midx1 = ks.MultiIndex.from_tuples([("a", "x"), ("b", "y"), ("c", "z")])
>>> midx2 = ks.MultiIndex.from_tuples([("c", "z"), ("d", "w")])
>>> midx1.intersection(midx2).sort_values()
MultiIndex([('c', 'z')],
           )
```

**databricks.koalas.MultiIndex.union**

`MultiIndex.union` (*other*, *sort=None*) → `databricks.koalas.indexes.base.Index`

Form the union of two Index objects.

**Parameters**

**other** [Index or array-like]

**sort** [bool or None, default None] Whether to sort the resulting Index.

**Returns**

**union** [Index]

**Examples****Index**

```
>>> idx1 = ks.Index([1, 2, 3, 4])
>>> idx2 = ks.Index([3, 4, 5, 6])
>>> idx1.union(idx2).sort_values()
Int64Index([1, 2, 3, 4, 5, 6], dtype='int64')
```

**MultiIndex**

```
>>> midx1 = ks.MultiIndex.from_tuples([("x", "a"), ("x", "b"), ("x", "c"), ("x",
↪ "d")])
>>> midx2 = ks.MultiIndex.from_tuples([("x", "c"), ("x", "d"), ("x", "e"), ("x",
↪ "f")])
>>> midx1.union(midx2).sort_values()
MultiIndex([('x', 'a'),
           ('x', 'b'),
```

(continues on next page)

(continued from previous page)

```

        ('x', 'c'),
        ('x', 'd'),
        ('x', 'e'),
        ('x', 'f')],
    )

```

### `databricks.koalas.MultiIndex.difference`

`MultiIndex.difference` (*other*, *sort=None*) → `databricks.koalas.indexes.base.Index`

Return a new Index with elements from the index that are not in *other*.

This is the set difference of two Index objects.

#### Parameters

**other** [Index or array-like]

**sort** [True or None, default None] Whether to sort the resulting index. \* True : Attempt to sort the result. \* None : Do not sort the result.

#### Returns

**difference** [Index]

### Examples

```

>>> idx1 = ks.Index([2, 1, 3, 4])
>>> idx2 = ks.Index([3, 4, 5, 6])
>>> idx1.difference(idx2, sort=True)
Int64Index([1, 2], dtype='int64')

```

### MultiIndex

```

>>> midx1 = ks.MultiIndex.from_tuples([('a', 'x', 1), ('b', 'y', 2), ('c', 'z', 3)])
>>> midx2 = ks.MultiIndex.from_tuples([('a', 'x', 1), ('b', 'z', 2), ('k', 'z', 3)])
>>> midx1.difference(midx2)
MultiIndex([('b', 'y', 2),
            ('c', 'z', 3)],
           )

```

### `databricks.koalas.MultiIndex.symmetric_difference`

`MultiIndex.symmetric_difference` (*other*, *result\_name=None*, *sort=None*) → `databricks.koalas.indexes.multi.MultiIndex`

Compute the symmetric difference of two MultiIndex objects.

#### Parameters

**other** [Index or array-like]

**result\_name** [list]

**sort** [True or None, default None] Whether to sort the resulting index. \* True : Attempt to sort the result. \* None : Do not sort the result.

**Returns****symmetric\_difference** [MultiIndex]**Notes**

`symmetric_difference` contains elements that appear in either `idx1` or `idx2` but not both. Equivalent to the Index created by `idx1.difference(idx2) | idx2.difference(idx1)` with duplicates dropped.

**Examples**

```
>>> midx1 = pd.MultiIndex([['lama', 'cow', 'falcon'],
...                        ['speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [0, 0, 0, 0, 1, 2, 0, 1, 2]])
>>> midx2 = pd.MultiIndex([['koalas', 'cow', 'falcon'],
...                        ['speed', 'weight', 'length']],
...                       [[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                       [0, 0, 0, 0, 1, 2, 0, 1, 2]])
>>> s1 = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3],
...                index=midx1)
>>> s2 = ks.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3],
...                index=midx2)
```

```
>>> s1.index.symmetric_difference(s2.index)
MultiIndex([('koalas', 'speed'),
            ( 'lama', 'speed')],
            )
```

You can set names of result Index.

```
>>> s1.index.symmetric_difference(s2.index, result_name=['a', 'b'])
MultiIndex([('koalas', 'speed'),
            ( 'lama', 'speed')],
            names=['a', 'b'])
```

You can set `sort` to `True`, if you want to sort the resulting index.

```
>>> s1.index.symmetric_difference(s2.index, sort=True)
MultiIndex([('koalas', 'speed'),
            ( 'lama', 'speed')],
            )
```

You can also use the `^` operator:

```
>>> s1.index ^ s2.index
MultiIndex([('koalas', 'speed'),
            ( 'lama', 'speed')],
            )
```

## MultilIndex Conversion

<code>MultiIndex.astype(dtype)</code>	Cast a Koalas object to a specified dtype dtype.
<code>MultiIndex.item()</code>	Return the first element of the underlying data as a python tuple.
<code>MultiIndex.to_list()</code>	Return a list of the values.
<code>MultiIndex.to_series([name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.
<code>MultiIndex.to_frame([index, name])</code>	Create a DataFrame with the levels of the MultiIndex as columns.
<code>MultiIndex.view()</code>	this is defined as a copy with the same identity
<code>MultiIndex.to_numpy([dtype, copy])</code>	A NumPy ndarray representing the values in this Index or MultiIndex.

### databricks.koalas.MultiIndex.astype

`MultiIndex.astype` (dtype: Union[str, type, numpy.dtype, pandas.core.dtypes.base.ExtensionDtype]) → Union[Index, Series]  
Cast a Koalas object to a specified dtype dtype.

#### Parameters

**dtype** [data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type.

#### Returns

**casted** [same type as caller]

#### See also:

[`to\_datetime`](#) Convert argument to datetime.

### Examples

```
>>> ser = ks.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
```

```
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

```
>>> ser.rename("a").to_frame().set_index("a").index.astype('int64')
Int64Index([1, 2], dtype='int64', name='a')
```

**databricks.koalas.MultiIndex.item**

`MultiIndex.item()` → `Tuple[Union[int, float, str, bytes, decimal.Decimal, datetime.date, None], ...]`

Return the first element of the underlying data as a python tuple.

**Returns**

**tuple** The first element of MultiIndex.

**Raises**

**ValueError** If the data is not length-1.

**Examples**

```
>>> kmidx = ks.MultiIndex.from_tuples([('a', 'x')])
>>> kmidx.item()
('a', 'x')
```

**databricks.koalas.MultiIndex.to\_list**

`MultiIndex.to_list()` → `List`

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

---

**Note:** This method should only be used if the resulting list is expected to be small, as all the data is loaded into the driver's memory.

---

**Examples****Index**

```
>>> idx = ks.Index([1, 2, 3, 4, 5])
>>> idx.to_list()
[1, 2, 3, 4, 5]
```

**MultiIndex**

```
>>> tuples = [(1, 'red'), (1, 'blue'), (2, 'red'), (2, 'green')]
>>> midx = ks.MultiIndex.from_tuples(tuples)
>>> midx.to_list()
[(1, 'red'), (1, 'blue'), (2, 'red'), (2, 'green')]
```

**databricks.koalas.MultiIndex.to\_series**

`MultiIndex.to_series` (*name: Union[Any, Tuple] = None*) → `databricks.koalas.series.Series`

Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.

**Parameters**

**name** [string, optional] name of resulting Series. If None, defaults to name of original index

**Returns**

**Series** [dtype will be based on the type of the Index values.]

**Examples**

```
>>> df = ks.DataFrame([(0.2, 0.3), (0.0, 0.6), (0.6, 0.0), (0.2, 0.1)],
...                   columns=['dogs', 'cats'],
...                   index=list('abcd'))
>>> df['dogs'].index.to_series()
a      a
b      b
c      c
d      d
dtype: object
```

**databricks.koalas.MultiIndex.to\_frame**

`MultiIndex.to_frame` (*index=True, name=None*) → `databricks.koalas.frame.DataFrame`

Create a DataFrame with the levels of the MultiIndex as columns. Column ordering is determined by the DataFrame constructor with data as a dict.

**Parameters**

**index** [boolean, default True] Set the index of the returned DataFrame as the original MultiIndex.

**name** [list / sequence of strings, optional] The passed names should substitute index level names.

**Returns**

**DataFrame** [a DataFrame containing the original MultiIndex data.]

See also:

[\*DataFrame\*](#)

## Examples

```
>>> tuples = [(1, 'red'), (1, 'blue'),
...           (2, 'red'), (2, 'blue')]
>>> idx = ks.MultiIndex.from_tuples(tuples, names=('number', 'color'))
>>> idx
MultiIndex([(1, 'red'),
            (1, 'blue'),
            (2, 'red'),
            (2, 'blue')],
           names=['number', 'color'])
>>> idx.to_frame()
      number color
number color
1      red      1  red
      blue      1  blue
2      red      2  red
      blue      2  blue
```

By default, the original Index is reused. To enforce a new Index:

```
>>> idx.to_frame(index=False)
      number color
0         1  red
1         1  blue
2         2  red
3         2  blue
```

To override the name of the resulting column, specify *name*:

```
>>> idx.to_frame(name=['n', 'c'])
      n  c
number color
1      red  1  red
      blue  1  blue
2      red  2  red
      blue  2  blue
```

## databricks.koalas.MultiIndex.view

`MultiIndex.view()` → `databricks.koalas.indexes.base.Index`  
 this is defined as a copy with the same identity

## databricks.koalas.MultiIndex.to\_numpy

`MultiIndex.to_numpy(dtype=None, copy=False)` → `numpy.ndarray`  
 A NumPy ndarray representing the values in this Index or MultiIndex.

---

**Note:** This method should only be used if the resulting NumPy ndarray is expected to be small, as all the data is loaded into the driver's memory.

---

## Parameters

**dtype** [str or numpy.dtype, optional] The dtype to pass to `numpy.asarray()`

**copy** [bool, default False] Whether to ensure that the returned value is a not a view on another array. Note that `copy=False` does not *ensure* that `to_numpy()` is no-copy. Rather, `copy=True` ensure that a copy is made, even if not strictly necessary.

#### Returns

**numpy.ndarray**

#### Examples

```
>>> ks.Series([1, 2, 3, 4]).index.to_numpy()
array([0, 1, 2, 3])
>>> ks.DataFrame({'a': ['a', 'b', 'c']}, index=[[1, 2, 3], [4, 5, 6]]).index.to_
↳numpy()
array([(1, 4), (2, 5), (3, 6)], dtype=object)
```

### 3.5.6 MultiIndex Spark-related

`MultiIndex.spark` provides features that does not exist in pandas but in Spark. These can be accessed by `MultiIndex.spark.<function/property>`.

<code>MultiIndex.spark.data_type</code>	Returns the data type as defined by Spark, as a Spark <code>DataType</code> object.
<code>MultiIndex.spark.column</code>	Spark Column object representing the Series/Index.
<code>MultiIndex.spark.transform(func)</code>	Applies a function that takes and returns a Spark column.

#### `databricks.koalas.MultiIndex.spark.data_type`

**property** `spark.data_type`

Returns the data type as defined by Spark, as a Spark `DataType` object.

#### `databricks.koalas.MultiIndex.spark.column`

**property** `spark.column`

Spark Column object representing the Series/Index.

---

**Note:** This Spark Column object is strictly stick to its base DataFrame the Series/Index was derived from.

---



**databricks.koalas.MultilIndex.spark.transform**

`spark.transform(func) → ks.Index`

Applies a function that takes and returns a Spark column. It allows to natively apply a Spark function and column APIs with the Spark column internally used in Series or Index. The output length of the Spark column should be same as input's.

---

**Note:** It requires to have the same input and output length; therefore, the aggregate Spark functions such as count does not work.

---

**Parameters**

**func** [function] Function to use for transforming the data by using Spark columns.

**Returns**

**Series or Index**

**Raises**

**ValueError** [If the output from the function is not a Spark column.]

**Examples**

```
>>> from pyspark.sql.functions import log
>>> df = ks.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]}, columns=["a", "b"])
>>> df
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.a.spark.transform(lambda c: log(c))
0    0.000000
1    0.693147
2    1.098612
Name: a, dtype: float64
```

```
>>> df.index.spark.transform(lambda c: c + 10)
Int64Index([10, 11, 12], dtype='int64')
```

```
>>> df.a.spark.transform(lambda c: c + df.b.spark.column)
0    5
1    7
2    9
Name: a, dtype: int64
```

## MultIndex Sorting

---

<code>MultiIndex.sort_values([ascending])</code>	Return a sorted copy of the index.
--	------------------------------------

---

### `databricks.koalas.MultiIndex.sort_values`

`MultiIndex.sort_values(ascending=True)` → `databricks.koalas.indexes.base.Index`  
 Return a sorted copy of the index.

---

**Note:** This method is not supported for pandas when index has NaN value. pandas raises unexpected `TypeError`, but we support treating NaN as the smallest value.

---

#### Parameters

**ascending** [bool, default True] Should the index values be sorted in an ascending order.

#### Returns

**sorted\_index** [ks.Index or ks.MultiIndex] Sorted copy of the index.

#### See also:

**`Series.sort_values`** Sort values of a Series.

**`DataFrame.sort_values`** Sort values in a DataFrame.

## Examples

```
>>> idx = ks.Index([10, 100, 1, 1000])
>>> idx
Int64Index([10, 100, 1, 1000], dtype='int64')
```

Sort values in ascending order (default behavior).

```
>>> idx.sort_values()
Int64Index([1, 10, 100, 1000], dtype='int64')
```

Sort values in descending order.

```
>>> idx.sort_values(ascending=False)
Int64Index([1000, 100, 10, 1], dtype='int64')
```

Support for MultiIndex.

```
>>> kidx = ks.MultiIndex.from_tuples([('a', 'x', 1), ('c', 'y', 2), ('b', 'z', 3)])
>>> kidx
MultiIndex([('a', 'x', 1),
            ('c', 'y', 2),
            ('b', 'z', 3)],
           )
```

```
>>> kidx.sort_values()
MultiIndex([('a', 'x', 1),
           ('b', 'z', 3),
           ('c', 'y', 2)],
          )
```

```
>>> kidx.sort_values(ascending=False)
MultiIndex([('c', 'y', 2),
           ('b', 'z', 3),
           ('a', 'x', 1)],
          )
```

### 3.5.7 DatetimeIndex

---

*DatetimeIndex*

Immutable ndarray-like of datetime64 data.

---

#### **databricks.koalas.DatetimeIndex**

**class** databricks.koalas.DatetimeIndex

Immutable ndarray-like of datetime64 data.

##### **Parameters**

**data** [array-like (1-dimensional), optional] Optional datetime-like data to construct index with.

**freq** [str or pandas offset object, optional] One of pandas date offset strings or corresponding objects. The string ‘infer’ can be passed in order to set the frequency of the index as the inferred frequency upon creation.

**normalize** [bool, default False] Normalize start/end dates to midnight before generating date range.

**closed** [{‘left’, ‘right’}, optional] Set whether to include *start* and *end* that are on the boundary. The default includes boundary points on either end.

**ambiguous** [‘infer’, bool-ndarray, ‘NaT’, default ‘raise’] When clocks moved backward due to DST, ambiguous times may arise. For example in Central European Time (UTC+01), when going from 03:00 DST to 02:00 non-DST, 02:30:00 local time occurs both at 00:30:00 UTC and at 01:30:00 UTC. In such a situation, the *ambiguous* parameter dictates how ambiguous times should be handled.

- ‘infer’ will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False signifies a non-DST time (note that this flag is only applicable for ambiguous times)
- ‘NaT’ will return NaT where there are ambiguous times
- ‘raise’ will raise an AmbiguousTimeError if there are ambiguous times.

**dayfirst** [bool, default False] If True, parse dates in *data* with the day first order.

**yearfirst** [bool, default False] If True parse dates in *data* with the year first order.

**dtype** [numpy.dtype or str, default None] Note that the only NumPy dtype allowed is ‘datetime64[ns]’.

**copy** [bool, default False] Make a copy of input ndarray.

**name** [label, default None] Name to be stored in the index.

**See also:**

**Index** The base pandas Index type.

**to\_datetime** Convert argument to datetime.

## Examples

```
>>> ks.DatetimeIndex(['1970-01-01', '1970-01-01', '1970-01-01'])
DatetimeIndex(['1970-01-01', '1970-01-01', '1970-01-01'], dtype='datetime64[ns]',
               freq=None)
```

From a Series:

```
>>> from datetime import datetime
>>> s = ks.Series([datetime(2021, 3, 1), datetime(2021, 3, 2)], index=[10, 20])
>>> ks.DatetimeIndex(s)
DatetimeIndex(['2021-03-01', '2021-03-02'], dtype='datetime64[ns]', freq=None)
```

From an Index:

```
>>> idx = ks.DatetimeIndex(['1970-01-01', '1970-01-01', '1970-01-01'])
>>> ks.DatetimeIndex(idx)
DatetimeIndex(['1970-01-01', '1970-01-01', '1970-01-01'], dtype='datetime64[ns]',
               freq=None)
```

**\_\_init\_\_** (\*args, \*\*kwargs)

Initialize self. See help(type(self)) for accurate signature.

## Methods

<code>__init__</code> (*args, **kwargs)	Initialize self.
<code>all</code> ([axis])	Return whether all elements are True.
<code>any</code> ([axis])	Return whether any element is True.
<code>append</code> (other)	Append a collection of Index options together.
<code>argmax</code> ()	Return a maximum argument indexer.
<code>argmin</code> ()	Return a minimum argument indexer.
<code>asof</code> (label)	Return the label from the index, or, if not present, the previous one.
<code>astype</code> (dtype)	Cast a Koalas object to a specified dtype dtype.
<code>ceil</code> (freq, *args, **kwargs)	Perform ceil operation on the data to the specified freq.
<code>copy</code> ([name, deep])	Make a copy of this object.
<code>day_name</code> ([locale])	Return the day names of the series with specified locale.
<code>delete</code> (loc)	Make new Index with passed location(-s) deleted.
<code>difference</code> (other[, sort])	Return a new Index with elements from the index that are not in other.
<code>drop</code> (labels)	Make new Index with passed list of labels deleted.
<code>drop_duplicates</code> ()	Return Index with duplicate values removed.

continues on next page

Table 94 – continued from previous page

<code>droplevel(level)</code>	Return index with requested level(s) removed.
<code>dropna()</code>	Return Index or MultiIndex without NA/NaN values
<code>equals(other)</code>	Determine if two Index objects contain the same elements.
<code>factorize([sort, na_sentinel])</code>	Encode the object as an enumerated type or categorical variable.
<code>fillna(value)</code>	Fill NA/NaN values with the specified value.
<code>floor(freq, *args, **kwargs)</code>	Perform floor operation on the data to the specified freq.
<code>get_level_values(level)</code>	Return Index if a valid level is given.
<code>holds_integer()</code>	Whether the type is an integer type.
<code>identical(other)</code>	Similar to equals, but check that other comparable attributes are also equal.
<code>indexer_at_time(time[, asof])</code>	Return index locations of values at particular time of day (e.g.
<code>indexer_between_time(start_time, end_time[, ...])</code>	Return index locations of values between particular times of day (e.g., 9:00-9:30AM).
<code>insert(loc, item)</code>	Make new Index inserting new item at location.
<code>intersection(other)</code>	Form the intersection of two Index objects.
<code>is_boolean()</code>	Return if the current index type is a boolean type.
<code>is_categorical()</code>	Return if the current index type is a categorical type.
<code>is_floating()</code>	Return if the current index type is a floating type.
<code>is_integer()</code>	Return if the current index type is an integer type.
<code>is_interval()</code>	Return if the current index type is an interval type.
<code>is_numeric()</code>	Return if the current index type is a numeric type.
<code>is_object()</code>	Return if the current index type is a object type.
<code>is_type_compatible(kind)</code>	Whether the index type is compatible with the provided type.
<code>isin(values)</code>	Check whether <i>values</i> are contained in Series or Index.
<code>isna()</code>	Detect existing (non-missing) values.
<code>isnull()</code>	Detect existing (non-missing) values.
<code>item()</code>	Return the first element of the underlying data as a python scalar.
<code>max()</code>	Return the maximum value of the Index.
<code>min()</code>	Return the minimum value of the Index.
<code>month_name([locale])</code>	Return the month names of the DatetimeIndex with specified locale.
<code>normalize()</code>	Convert times to midnight.
<code>notna()</code>	Detect existing (non-missing) values.
<code>notnull()</code>	Detect existing (non-missing) values.
<code>nunique([dropna, approx, rsd])</code>	Return number of unique elements in the object.
<code>rename(name[, inplace])</code>	Alter Index or MultiIndex name.
<code>repeat(repeats)</code>	Repeat elements of a Index/MultiIndex.
<code>round(freq, *args, **kwargs)</code>	Perform round operation on the data to the specified freq.
<code>set_names(names[, level, inplace])</code>	Set Index or MultiIndex name.
<code>shift([periods, fill_value])</code>	Shift Series/Index by desired number of periods.
<code>sort(*args, **kwargs)</code>	Use sort_values instead.
<code>sort_values([ascending])</code>	Return a sorted copy of the index.

continues on next page

Table 94 – continued from previous page

<code>strftime(date_format)</code>	Convert to a string Index using specified <code>date_format</code> .
<code>symmetric_difference(other[, result_name, sort])</code>	Compute the symmetric difference of two Index objects.
<code>take(indices)</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>toPandas()</code>	Return a pandas Index.
<code>to_frame([index, name])</code>	Create a DataFrame with a column containing the Index.
<code>to_list()</code>	Return a list of the values.
<code>to_numpy([dtype, copy])</code>	A NumPy ndarray representing the values in this Index or MultiIndex.
<code>to_pandas()</code>	Return a pandas Index.
<code>to_series([name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index.
<code>tolist()</code>	Return a list of the values.
<code>transpose()</code>	Return the transpose, For index, It will be index itself.
<code>union(other[, sort])</code>	Form the union of two Index objects.
<code>unique([level])</code>	Return unique values in the index.
<code>value_counts([normalize, sort, ascending, ...])</code>	Return a Series containing counts of unique values.
<code>view()</code>	this is defined as a copy with the same identity

### Attributes

<code>T</code>	Return the transpose, For index, It will be index itself.
<code>asi8</code>	Integer representation of the values.
<code>day</code>	The days of the datetime.
<code>day_of_week</code>	The day of the week with Monday=0, Sunday=6.
<code>day_of_year</code>	The ordinal day of the year.
<code>dayofweek</code>	The day of the week with Monday=0, Sunday=6.
<code>dayofyear</code>	The ordinal day of the year.
<code>days_in_month</code>	The number of days in the month.
<code>daysinmonth</code>	The number of days in the month.
<code>dtype</code>	Return the dtype object of the underlying data.
<code>empty</code>	Returns true if the current object is empty.
<code>has_duplicates</code>	If index has duplicates, return True, otherwise False.
<code>hasnans</code>	Return True if it has any missing values.
<code>hour</code>	The hours of the datetime.
<code>inferred_type</code>	Return a string of the type inferred from the values.
<code>is_all_dates</code>	Return if all data types of the index are datetime.
<code>is_leap_year</code>	Boolean indicator if the date belongs to a leap year.
<code>is_monotonic</code>	Return boolean if values in the object are monotonically increasing.
<code>is_monotonic_decreasing</code>	Return boolean if values in the object are monotonically decreasing.

continues on next page

Table 95 – continued from previous page

<code>is_monotonic_increasing</code>	Return boolean if values in the object are monotonically increasing.
<code>is_month_end</code>	Indicates whether the date is the last day of the month.
<code>is_month_start</code>	Indicates whether the date is the first day of the month.
<code>is_quarter_end</code>	Indicator for whether the date is the last day of a quarter.
<code>is_quarter_start</code>	Indicator for whether the date is the first day of a quarter.
<code>is_unique</code>	Return if the index has unique values.
<code>is_year_end</code>	Indicate whether the date is the last day of the year.
<code>is_year_start</code>	Indicate whether the date is the first day of a year.
<code>microsecond</code>	The microseconds of the datetime.
<code>minute</code>	The minutes of the datetime.
<code>month</code>	The month of the timestamp as January = 1 December = 12.
<code>name</code>	Return name of the Index.
<code>names</code>	Return names of the Index.
<code>ndim</code>	Return an int representing the number of array dimensions.
<code>nlevels</code>	Number of levels in Index & MultiIndex.
<code>quarter</code>	The quarter of the date.
<code>second</code>	The seconds of the datetime.
<code>shape</code>	Return a tuple of the shape of the underlying data.
<code>size</code>	Return an int representing the number of elements in this object.
<code>spark_column</code>	Spark Column object representing the Series/Index.
<code>spark_type</code>	Returns the data type as defined by Spark, as a Spark DataType object.
<code>values</code>	Return an array representing the data in the Index.
<code>week</code>	The week ordinal of the year.
<code>weekday</code>	The day of the week with Monday=0, Sunday=6.
<code>weekofyear</code>	The week ordinal of the year.
<code>year</code>	The year of the datetime.

### Time/date components

<code>DatetimeIndex.year</code>	The year of the datetime.
<code>DatetimeIndex.month</code>	The month of the timestamp as January = 1 December = 12.
<code>DatetimeIndex.day</code>	The days of the datetime.
<code>DatetimeIndex.hour</code>	The hours of the datetime.
<code>DatetimeIndex.minute</code>	The minutes of the datetime.
<code>DatetimeIndex.second</code>	The seconds of the datetime.
<code>DatetimeIndex.microsecond</code>	The microseconds of the datetime.
<code>DatetimeIndex.week</code>	The week ordinal of the year.
<code>DatetimeIndex.weekofyear</code>	The week ordinal of the year.
<code>DatetimeIndex.dayofweek</code>	The day of the week with Monday=0, Sunday=6.

continues on next page

Table 96 – continued from previous page

<i>DatetimeIndex.day_of_week</i>	The day of the week with Monday=0, Sunday=6.
<i>DatetimeIndex.weekday</i>	The day of the week with Monday=0, Sunday=6.
<i>DatetimeIndex.dayofyear</i>	The ordinal day of the year.
<i>DatetimeIndex.day_of_year</i>	The ordinal day of the year.
<i>DatetimeIndex.quarter</i>	The quarter of the date.
<i>DatetimeIndex.is_month_start</i>	Indicates whether the date is the first day of the month.
<i>DatetimeIndex.is_month_end</i>	Indicates whether the date is the last day of the month.
<i>DatetimeIndex.is_quarter_start</i>	Indicator for whether the date is the first day of a quarter.
<i>DatetimeIndex.is_quarter_end</i>	Indicator for whether the date is the last day of a quarter.
<i>DatetimeIndex.is_year_start</i>	Indicate whether the date is the first day of a year.
<i>DatetimeIndex.is_year_end</i>	Indicate whether the date is the last day of the year.
<i>DatetimeIndex.is_leap_year</i>	Boolean indicator if the date belongs to a leap year.
<i>DatetimeIndex.daysinmonth</i>	The number of days in the month.
<i>DatetimeIndex.days_in_month</i>	The number of days in the month.

**databricks.koalas.DatetimeIndex.year****property** `DatetimeIndex.year`

The year of the datetime.

**databricks.koalas.DatetimeIndex.month****property** `DatetimeIndex.month`

The month of the timestamp as January = 1 December = 12.

**databricks.koalas.DatetimeIndex.day****property** `DatetimeIndex.day`

The days of the datetime.

**databricks.koalas.DatetimeIndex.hour****property** `DatetimeIndex.hour`

The hours of the datetime.

**databricks.koalas.DatetimeIndex.minute****property** `DatetimeIndex.minute`

The minutes of the datetime.



**databricks.koalas.DatetimeIndex.second**

**property** `DatetimeIndex.second`  
The seconds of the datetime.

**databricks.koalas.DatetimeIndex.microsecond**

**property** `DatetimeIndex.microsecond`  
The microseconds of the datetime.

**databricks.koalas.DatetimeIndex.week**

**property** `DatetimeIndex.week`  
The week ordinal of the year.

**databricks.koalas.DatetimeIndex.weekofyear**

**property** `DatetimeIndex.weekofyear`  
The week ordinal of the year.

**databricks.koalas.DatetimeIndex.dayofweek**

**property** `DatetimeIndex.dayofweek`  
The day of the week with Monday=0, Sunday=6. Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor) or DatetimeIndex.

**Returns**

**Series or Index** Containing integers indicating the day number.

**See also:**

*Series.dt.dayofweek* Alias.

*Series.dt.weekday* Alias.

*Series.dt.day\_name* Returns the name of the day of the week.

**Examples**

```
>>> idx = ks.date_range('2016-12-31', '2017-01-08', freq='D')
>>> idx.dayofweek
Int64Index([5, 6, 0, 1, 2, 3, 4, 5, 6], dtype='int64')
```

**databricks.koalas.DatetimeIndex.day\_of\_week****property** `DatetimeIndex.day_of_week`

The day of the week with Monday=0, Sunday=6. Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor) or DatetimeIndex.

**Returns**

**Series or Index** Containing integers indicating the day number.

**See also:**

*`Series.dt.dayofweek`* Alias.

*`Series.dt.weekday`* Alias.

*`Series.dt.day_name`* Returns the name of the day of the week.

**Examples**

```
>>> idx = ks.date_range('2016-12-31', '2017-01-08', freq='D')
>>> idx.dayofweek
Int64Index([5, 6, 0, 1, 2, 3, 4, 5, 6], dtype='int64')
```

**databricks.koalas.DatetimeIndex.weekday****property** `DatetimeIndex.weekday`

The day of the week with Monday=0, Sunday=6. Return the day of the week. It is assumed the week starts on Monday, which is denoted by 0 and ends on Sunday which is denoted by 6. This method is available on both Series with datetime values (using the *dt* accessor) or DatetimeIndex.

**Returns**

**Series or Index** Containing integers indicating the day number.

**See also:**

*`Series.dt.dayofweek`* Alias.

*`Series.dt.weekday`* Alias.

*`Series.dt.day_name`* Returns the name of the day of the week.

**Examples**

```
>>> idx = ks.date_range('2016-12-31', '2017-01-08', freq='D')
>>> idx.dayofweek
Int64Index([5, 6, 0, 1, 2, 3, 4, 5, 6], dtype='int64')
```

**databricks.koalas.DatetimeIndex.dayofyear****property** `DatetimeIndex.dayofyear`

The ordinal day of the year.

**databricks.koalas.DatetimeIndex.day\_of\_year****property** `DatetimeIndex.day_of_year`

The ordinal day of the year.

**databricks.koalas.DatetimeIndex.quarter****property** `DatetimeIndex.quarter`

The quarter of the date.

**databricks.koalas.DatetimeIndex.is\_month\_start****property** `DatetimeIndex.is_month_start`

Indicates whether the date is the first day of the month.

**Returns****Index** Returns a Index with boolean values**See also:***`is_month_end`* Return a boolean indicating whether the date is the last day of the month.**Examples**

```
>>> idx = ks.date_range("2018-02-27", periods=3)
>>> idx.is_month_start
Index([False, False, True], dtype='object')
```

**databricks.koalas.DatetimeIndex.is\_month\_end****property** `DatetimeIndex.is_month_end`

Indicates whether the date is the last day of the month.

**Returns****Index** Returns a Index with boolean values.**See also:***`is_month_start`* Return a boolean indicating whether the date is the first day of the month.

## Examples

```
>>> idx = ks.date_range("2018-02-27", periods=3)
>>> idx.is_month_end
Index([False, True, False], dtype='object')
```

### `databricks.koalas.DatetimeIndex.is_quarter_start`

**property** `DatetimeIndex.is_quarter_start`

Indicator for whether the date is the first day of a quarter.

#### Returns

**is\_quarter\_start** [Index] Returns an Index with boolean values.

See also:

[`quarter`](#) Return the quarter of the date.

[`is\_quarter\_end`](#) Similar property for indicating the quarter start.

## Examples

```
>>> idx = ks.date_range('2017-03-30', periods=4)
>>> idx.is_quarter_start
Index([False, False, True, False], dtype='object')
```

### `databricks.koalas.DatetimeIndex.is_quarter_end`

**property** `DatetimeIndex.is_quarter_end`

Indicator for whether the date is the last day of a quarter.

#### Returns

**is\_quarter\_end** [Index] Returns an Index with boolean values.

See also:

[`quarter`](#) Return the quarter of the date.

[`is\_quarter\_start`](#) Similar property indicating the quarter start.

## Examples

```
>>> idx = ks.date_range('2017-03-30', periods=4)
>>> idx.is_quarter_end
Index([False, True, False, False], dtype='object')
```

**databricks.koalas.DatetimeIndex.is\_year\_start**

**property** `DatetimeIndex.is_year_start`  
 Indicate whether the date is the first day of a year.

**Returns**

**Index** Returns an Index with boolean values.

**See also:**

*[is\\_year\\_end](#)* Similar property indicating the last day of the year.

**Examples**

```
>>> idx = ks.date_range("2017-12-30", periods=3)
>>> idx.is_year_start
Index([False, False, True], dtype='object')
```

**databricks.koalas.DatetimeIndex.is\_year\_end**

**property** `DatetimeIndex.is_year_end`  
 Indicate whether the date is the last day of the year.

**Returns**

**Index** Returns an Index with boolean values.

**See also:**

*[is\\_year\\_start](#)* Similar property indicating the start of the year.

**Examples**

```
>>> idx = ks.date_range("2017-12-30", periods=3)
>>> idx.is_year_end
Index([False, True, False], dtype='object')
```

**databricks.koalas.DatetimeIndex.is\_leap\_year**

**property** `DatetimeIndex.is_leap_year`  
 Boolean indicator if the date belongs to a leap year.

A leap year is a year, which has 366 days (instead of 365) including 29th of February as an intercalary day. Leap years are years which are multiples of four with the exception of years divisible by 100 but not by 400.

**Returns**

**Index** Booleans indicating if dates belong to a leap year.

## Examples

```
>>> idx = ks.date_range("2012-01-01", "2015-01-01", freq="Y")
>>> idx.is_leap_year
Index([True, False, False], dtype='object')
```

### `databricks.koalas.DatetimeIndex.daysinmonth`

**property** `DatetimeIndex.daysinmonth`

The number of days in the month.

### `databricks.koalas.DatetimeIndex.days_in_month`

**property** `DatetimeIndex.days_in_month`

The number of days in the month.

## Selecting

<code>DatetimeIndex.indexer_between_time(...[, ...])</code>	Return index locations of values between particular times of day (e.g., 9:00-9:30AM).
<code>DatetimeIndex.indexer_at_time(time[, asof])</code>	Return index locations of values at particular time of day (e.g.

### `databricks.koalas.DatetimeIndex.indexer_between_time`

`DatetimeIndex.indexer_between_time` (*start\_time: Union[datetime.time, str], end\_time: Union[datetime.time, str], include\_start: bool = True, include\_end: bool = True*) → `databricks.koalas.indexes.base.Index`  
Return index locations of values between particular times of day (e.g., 9:00-9:30AM).

#### Parameters

**start\_time, end\_time** [datetime.time, str] Time passed either as object (datetime.time) or as string in appropriate format (“%H:%M”, “%H%M”, “%I:%M%p”, “%I%M%p”, “%H:%M:%S”, “%H%M%S”, “%I:%M:%S%p”, “%I%M%S%p”).

**include\_start** [bool, default True]

**include\_end** [bool, default True]

#### Returns

**values\_between\_time** [Index of integers]

## Examples

```
>>> kidx = ks.date_range("2000-01-01", periods=3, freq="T")
>>> kidx
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 00:01:00',
              '2000-01-01 00:02:00'],
              dtype='datetime64[ns]', freq=None)
```

```
>>> kidx.indexer_between_time("00:01", "00:02").sort_values()
Int64Index([1, 2], dtype='int64')
```

```
>>> kidx.indexer_between_time("00:01", "00:02", include_end=False)
Int64Index([1], dtype='int64')
```

```
>>> kidx.indexer_between_time("00:01", "00:02", include_start=False)
Int64Index([2], dtype='int64')
```

## databricks.koalas.DatetimeIndex.indexer\_at\_time

`DatetimeIndex.indexer_at_time` (*time*: `Union[datetime.time, str]`, *asof*: `bool = False`) → `databricks.koalas.indexes.base.Index`  
 Return index locations of values at particular time of day (e.g. 9:30AM).

### Parameters

**time** [`datetime.time` or `str`] Time passed in either as object (`datetime.time`) or as string in appropriate format (“%H:%M”, “%H%M”, “%I:%M%p”, “%I%M%p”, “%H:%M:%S”, “%H%M%S”, “%I:%M:%S%p”, “%I%M%S%p”).

### Returns

**values\_at\_time** [Index of integers]

## Examples

```
>>> kidx = ks.date_range("2000-01-01", periods=3, freq="T")
>>> kidx
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 00:01:00',
              '2000-01-01 00:02:00'],
              dtype='datetime64[ns]', freq=None)
```

```
>>> kidx.indexer_at_time("00:00")
Int64Index([0], dtype='int64')
```

```
>>> kidx.indexer_at_time("00:01")
Int64Index([1], dtype='int64')
```

## Time-specific operations

<code>DatetimeIndex.normalize()</code>	Convert times to midnight.
<code>DatetimeIndex.strftime(date_format)</code>	Convert to a string Index using specified date_format.
<code>DatetimeIndex.round(freq, *args, **kwargs)</code>	Perform round operation on the data to the specified freq.
<code>DatetimeIndex.floor(freq, *args, **kwargs)</code>	Perform floor operation on the data to the specified freq.
<code>DatetimeIndex.ceil(freq, *args, **kwargs)</code>	Perform ceil operation on the data to the specified freq.
<code>DatetimeIndex.month_name([locale])</code>	Return the month names of the DatetimeIndex with specified locale.
<code>DatetimeIndex.day_name([locale])</code>	Return the day names of the series with specified locale.

### `databricks.koalas.DatetimeIndex.normalize`

`DatetimeIndex.normalize()` → `databricks.koalas.indexes.datetimes.DatetimeIndex`

Convert times to midnight.

The time component of the date-time is converted to midnight i.e. 00:00:00. This is useful in cases, when the time does not matter. Length is unaltered. The timezones are unaffected.

This method is available on Series with datetime values under the `.dt` accessor.

#### Returns

**DatetimeIndex** The same type as the original data.

See also:

**floor** Floor the series to the specified freq.

**ceil** Ceil the series to the specified freq.

**round** Round the series to the specified freq.

#### Examples

```
>>> idx = ks.date_range(start='2014-08-01 10:00', freq='H', periods=3)
>>> idx.normalize()
DatetimeIndex(['2014-08-01', '2014-08-01', '2014-08-01'], dtype='datetime64[ns]',
               freq=None)
```

### `databricks.koalas.DatetimeIndex.strftime`

`DatetimeIndex.strftime(date_format: str)` → `databricks.koalas.indexes.base.Index`

Convert to a string Index using specified date\_format.

Return an Index of formatted strings specified by date\_format, which supports the same string format as the python standard library. Details of the string format can be found in python string format doc.

#### Parameters

**date\_format** [str] Date format string (e.g. “%%Y-%%m-%%d”).

#### Returns



**Index** Index of formatted strings.

**See also:**

**`normalize`** Return series with times to midnight.

**`round`** Round the series to the specified freq.

**`floor`** Floor the series to the specified freq.

## Examples

```
>>> idx = ks.date_range(pd.Timestamp("2018-03-10 09:00"), periods=3, freq='s')
>>> idx.strftime('%B %d, %Y, %r')
Index(['March 10, 2018, 09:00:00 AM', 'March 10, 2018, 09:00:01 AM',
      'March 10, 2018, 09:00:02 AM'],
      dtype='object')
```

## **databricks.koalas.DatetimeIndex.round**

`DatetimeIndex.round(freq, *args, **kwargs)` → `databricks.koalas.indexes.datetimes.DatetimeIndex`  
 Perform round operation on the data to the specified freq.

### Parameters

**freq** [str or Offset] The frequency level to round the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end).

### Returns

**DatetimeIndex**

### Raises

**ValueError** if the *freq* cannot be converted.

## Examples

```
>>> rng = ks.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng.round("H")
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
              '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

## **databricks.koalas.DatetimeIndex.floor**

`DatetimeIndex.floor(freq, *args, **kwargs)` → `databricks.koalas.indexes.datetimes.DatetimeIndex`  
 Perform floor operation on the data to the specified freq.

### Parameters

**freq** [str or Offset] The frequency level to floor the index to. Must be a fixed frequency like 'S' (second) not 'ME' (month end).

### Returns

**DatetimeIndex****Raises**

**ValueError** if the *freq* cannot be converted.

**Examples**

```
>>> rng = ks.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng.floor("H")
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
              '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

**databricks.koalas.DatetimeIndex.ceil**

`DatetimeIndex.ceil(freq, *args, **kwargs)` → `databricks.koalas.indexes.datetimes.DatetimeIndex`

Perform ceil operation on the data to the specified freq.

**Parameters**

**freq** [str or Offset] The frequency level to ceil the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end).

**Returns**

**DatetimeIndex**

**Raises**

**ValueError** if the *freq* cannot be converted.

**Examples**

```
>>> rng = ks.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng.ceil('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
              '2018-01-01 13:00:00'],
              dtype='datetime64[ns]', freq=None)
```

**databricks.koalas.DatetimeIndex.month\_name**

`DatetimeIndex.month_name(locale: Optional[str] = None)` → `databricks.koalas.indexes.base.Index`

Return the month names of the DatetimeIndex with specified locale.

**Parameters**

**locale** [str, optional] Locale determining the language in which to return the month name. Default is English locale.

**Returns**

**Index** Index of month names.

## Examples

```
>>> idx = ks.date_range(start='2018-01', freq='M', periods=3)
>>> idx.month_name()
Index(['January', 'February', 'March'], dtype='object')
```

### `databricks.koalas.DatetimeIndex.day_name`

`DatetimeIndex.day_name` (*locale: Optional[str] = None*) → `databricks.koalas.indexes.base.Index`

Return the day names of the series with specified locale.

#### Parameters

**locale** [str, optional] Locale determining the language in which to return the day name. Default is English locale.

#### Returns

**Index** Index of day names.

## Examples

```
>>> idx = ks.date_range(start='2018-01-01', freq='D', periods=3)
>>> idx.day_name()
Index(['Monday', 'Tuesday', 'Wednesday'], dtype='object')
```

## 3.6 Window

Rolling objects are returned by `.rolling` calls: `koalas.DataFrame.rolling()`, `koalas.Series.rolling()`, etc. Expanding objects are returned by `.expanding` calls: `koalas.DataFrame.expanding()`, `koalas.Series.expanding()`, etc.

### 3.6.1 Standard moving window functions

<code>Rolling.count()</code>	The rolling count of any non-NaN observations inside the window.
<code>Rolling.sum()</code>	Calculate rolling summation of given DataFrame or Series.
<code>Rolling.min()</code>	Calculate the rolling minimum.
<code>Rolling.max()</code>	Calculate the rolling maximum.
<code>Rolling.mean()</code>	Calculate the rolling mean of the values.

**databricks.koalas.window.Rolling.count**

`Rolling.count()` → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]

The rolling count of any non-NaN observations inside the window.

---

**Note:** the current implementation of this API uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

**Returns**

**Series.expanding** [Calling object with Series data.]

**DataFrame.expanding** [Calling object with DataFrames.]

**Series.count** [Count of the full Series.]

**DataFrame.count** [Count of the full DataFrame.]

**Examples**

```
>>> s = ks.Series([2, 3, float("nan"), 10])
>>> s.rolling(1).count()
0    1.0
1    1.0
2    0.0
3    1.0
dtype: float64
```

```
>>> s.rolling(3).count()
0    1.0
1    2.0
2    2.0
3    2.0
dtype: float64
```

```
>>> s.to_frame().rolling(1).count()
0
0    1.0
1    1.0
2    0.0
3    1.0
```

```
>>> s.to_frame().rolling(3).count()
0
0    1.0
1    2.0
2    2.0
3    2.0
```

**databricks.koalas.window.Rolling.sum**

`Rolling.sum()` → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]

Calculate rolling summation of given DataFrame or Series.

**Note:** the current implementation of this API uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

**Returns**

**Series or DataFrame** Same type as the input, with the same index, containing the rolling summation.

**See also:**

**Series.expanding** Calling object with Series data.

**DataFrame.expanding** Calling object with DataFrames.

**Series.sum** Reducing sum for Series.

**DataFrame.sum** Reducing sum for DataFrame.

**Examples**

```
>>> s = ks.Series([4, 3, 5, 2, 6])
>>> s
0    4
1    3
2    5
3    2
4    6
dtype: int64
```

```
>>> s.rolling(2).sum()
0    NaN
1    7.0
2    8.0
3    7.0
4    8.0
dtype: float64
```

```
>>> s.rolling(3).sum()
0    NaN
1    NaN
2    12.0
3    10.0
4    13.0
dtype: float64
```

For DataFrame, each rolling summation is computed column-wise.

```
>>> df = ks.DataFrame({"A": s.to_numpy(), "B": s.to_numpy() ** 2})
>>> df
   A  B
0  4 16
1  3  9
2  5 25
3  2  4
4  6 36
```

```
>>> df.rolling(2).sum()
   A  B
0 NaN NaN
1 7.0 25.0
2 8.0 34.0
3 7.0 29.0
4 8.0 40.0
```

```
>>> df.rolling(3).sum()
   A  B
0 NaN NaN
1 NaN NaN
2 12.0 50.0
3 10.0 38.0
4 13.0 65.0
```

### **databricks.koalas.window.Rolling.min**

`Rolling.min()` → `Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]`  
Calculate the rolling minimum.

---

**Note:** the current implementation of this API uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

#### **Returns**

**Series or DataFrame** Returned object type is determined by the caller of the rolling calculation.

#### **See also:**

**Series.rolling** Calling object with a Series.

**DataFrame.rolling** Calling object with a DataFrame.

**Series.min** Similar method for Series.

**DataFrame.min** Similar method for DataFrame.

## Examples

```
>>> s = ks.Series([4, 3, 5, 2, 6])
>>> s
0    4
1    3
2    5
3    2
4    6
dtype: int64
```

```
>>> s.rolling(2).min()
0    NaN
1    3.0
2    3.0
3    2.0
4    2.0
dtype: float64
```

```
>>> s.rolling(3).min()
0    NaN
1    NaN
2    3.0
3    2.0
4    2.0
dtype: float64
```

For DataFrame, each rolling minimum is computed column-wise.

```
>>> df = ks.DataFrame({"A": s.to_numpy(), "B": s.to_numpy() ** 2})
>>> df
   A  B
0  4  16
1  3   9
2  5  25
3  2   4
4  6  36
```

```
>>> df.rolling(2).min()
   A  B
0 NaN NaN
1 3.0 9.0
2 3.0 9.0
3 2.0 4.0
4 2.0 4.0
```

```
>>> df.rolling(3).min()
   A  B
0 NaN NaN
1 NaN NaN
2 3.0 9.0
3 2.0 4.0
4 2.0 4.0
```

**databricks.koalas.window.Rolling.max**

`Rolling.max()` → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]  
Calculate the rolling maximum.

---

**Note:** the current implementation of this API uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

**Returns**

**Series or DataFrame** Return type is determined by the caller.

**See also:**

**Series.rolling** Series rolling.

**DataFrame.rolling** DataFrame rolling.

**Series.max** Similar method for Series.

**DataFrame.max** Similar method for DataFrame.

**Examples**

```
>>> s = ks.Series([4, 3, 5, 2, 6])
>>> s
0    4
1    3
2    5
3    2
4    6
dtype: int64
```

```
>>> s.rolling(2).max()
0    NaN
1    4.0
2    5.0
3    5.0
4    6.0
dtype: float64
```

```
>>> s.rolling(3).max()
0    NaN
1    NaN
2    5.0
3    5.0
4    6.0
dtype: float64
```

For DataFrame, each rolling maximum is computed column-wise.



```
>>> df = ks.DataFrame({"A": s.to_numpy(), "B": s.to_numpy() ** 2})
>>> df
```

	A	B
0	4	16
1	3	9
2	5	25
3	2	4
4	6	36

```
>>> df.rolling(2).max()
```

	A	B
0	NaN	NaN
1	4.0	16.0
2	5.0	25.0
3	5.0	25.0
4	6.0	36.0

```
>>> df.rolling(3).max()
```

	A	B
0	NaN	NaN
1	NaN	NaN
2	5.0	25.0
3	5.0	25.0
4	6.0	36.0

### `databricks.koalas.window.Rolling.mean`

`Rolling.mean()` → `Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]`  
 Calculate the rolling mean of the values.

---

**Note:** the current implementation of this API uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

#### Returns

**Series or DataFrame** Returned object type is determined by the caller of the rolling calculation.

#### See also:

**Series.rolling** Calling object with Series data.

**DataFrame.rolling** Calling object with DataFrames.

**Series.mean** Equivalent method for Series.

**DataFrame.mean** Equivalent method for DataFrame.

## Examples

```
>>> s = ks.Series([4, 3, 5, 2, 6])
>>> s
0      4
1      3
2      5
3      2
4      6
dtype: int64
```

```
>>> s.rolling(2).mean()
0      NaN
1      3.5
2      4.0
3      3.5
4      4.0
dtype: float64
```

```
>>> s.rolling(3).mean()
0      NaN
1      NaN
2      4.000000
3      3.333333
4      4.333333
dtype: float64
```

For DataFrame, each rolling mean is computed column-wise.

```
>>> df = ks.DataFrame({"A": s.to_numpy(), "B": s.to_numpy() ** 2})
>>> df
   A  B
0  4  16
1  3   9
2  5  25
3  2   4
4  6  36
```

```
>>> df.rolling(2).mean()
   A    B
0 NaN NaN
1 3.5 12.5
2 4.0 17.0
3 3.5 14.5
4 4.0 20.0
```

```
>>> df.rolling(3).mean()
   A    B
0 NaN NaN
1 NaN NaN
2 4.000000 16.666667
3 3.333333 12.666667
4 4.333333 21.666667
```

### 3.6.2 Standard expanding window functions

<code>Expanding.count()</code>	The expanding count of any non-NaN observations inside the window.
<code>Expanding.sum()</code>	Calculate expanding summation of given DataFrame or Series.
<code>Expanding.min()</code>	Calculate the expanding minimum.
<code>Expanding.max()</code>	Calculate the expanding maximum.
<code>Expanding.mean()</code>	Calculate the expanding mean of the values.

#### `databricks.koalas.window.Expanding.count`

`Expanding.count()` → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]

The expanding count of any non-NaN observations inside the window.

**Note:** the current implementation of this API uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

#### Returns

**Series or DataFrame** Returned object type is determined by the caller of the expanding calculation.

#### See also:

**Series.expanding** Calling object with Series data.

**DataFrame.expanding** Calling object with DataFrames.

**Series.count** Count of the full Series.

**DataFrame.count** Count of the full DataFrame.

#### Examples

```
>>> s = ks.Series([2, 3, float("nan"), 10])
>>> s.expanding().count()
0    1.0
1    2.0
2    2.0
3    3.0
dtype: float64
```

```
>>> s.to_frame().expanding().count()
0
0    1.0
1    2.0
2    2.0
3    3.0
```

## `databricks.koalas.window.Expanding.sum`

`Expanding.sum()` → `Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]`

Calculate expanding summation of given DataFrame or Series.

---

**Note:** the current implementation of this API uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

### Returns

**Series or DataFrame** Same type as the input, with the same index, containing the expanding summation.

See also:

**Series.expanding** Calling object with Series data.

**DataFrame.expanding** Calling object with DataFrames.

**Series.sum** Reducing sum for Series.

**DataFrame.sum** Reducing sum for DataFrame.

### Examples

```
>>> s = ks.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> s.expanding(3).sum()
0    NaN
1    NaN
2    6.0
3   10.0
4   15.0
dtype: float64
```

For DataFrame, each expanding summation is computed column-wise.

```
>>> df = ks.DataFrame({"A": s.to_numpy(), "B": s.to_numpy() ** 2})
>>> df
   A  B
0  1  1
1  2  4
2  3  9
3  4 16
4  5 25
```

```
>>> df.expanding(3).sum()
      A      B
0   NaN   NaN
1   NaN   NaN
2   6.0  14.0
3  10.0  30.0
4  15.0  55.0
```

### `databricks.koalas.window.Expanding.min`

`Expanding.min()` → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]

Calculate the expanding minimum.

**Note:** the current implementation of this API uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

#### Returns

**Series or DataFrame** Returned object type is determined by the caller of the expanding calculation.

#### See also:

**Series.expanding** Calling object with a Series.

**DataFrame.expanding** Calling object with a DataFrame.

**Series.min** Similar method for Series.

**DataFrame.min** Similar method for DataFrame.

### Examples

Performing a expanding minimum with a window size of 3.

```
>>> s = ks.Series([4, 3, 5, 2, 6])
>>> s.expanding(3).min()
0   NaN
1   NaN
2   3.0
3   2.0
4   2.0
dtype: float64
```

### `databricks.koalas.window.Expanding.max`

`Expanding.max()` → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]  
Calculate the expanding maximum.

---

**Note:** the current implementation of this API uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

#### Returns

**Series or DataFrame** Return type is determined by the caller.

#### See also:

**Series.expanding** Calling object with Series data.

**DataFrame.expanding** Calling object with DataFrames.

**Series.max** Similar method for Series.

**DataFrame.max** Similar method for DataFrame.

### Examples

Performing a expanding minimum with a window size of 3.

```
>>> s = ks.Series([4, 3, 5, 2, 6])
>>> s.expanding(3).max()
0      NaN
1      NaN
2      5.0
3      5.0
4      6.0
dtype: float64
```

### `databricks.koalas.window.Expanding.mean`

`Expanding.mean()` → Union[databricks.koalas.series.Series, databricks.koalas.frame.DataFrame]  
Calculate the expanding mean of the values.

---

**Note:** the current implementation of this API uses Spark's Window without specifying partition specification. This leads to move all data into single partition in single machine and could cause serious performance degradation. Avoid this method against very large dataset.

---

#### Returns

**Series or DataFrame** Returned object type is determined by the caller of the expanding calculation.

#### See also:

**Series.expanding** Calling object with Series data.

**DataFrame.expanding** Calling object with DataFrames.

**Series.mean** Equivalent method for Series.

**DataFrame.mean** Equivalent method for DataFrame.

## Examples

The below examples will show expanding mean calculations with window sizes of two and three, respectively.

```
>>> s = ks.Series([1, 2, 3, 4])
>>> s.expanding(2).mean()
0      NaN
1      1.5
2      2.0
3      2.5
dtype: float64
```

```
>>> s.expanding(3).mean()
0      NaN
1      NaN
2      2.0
3      2.5
dtype: float64
```

## 3.7 GroupBy

GroupBy objects are returned by groupby calls: `DataFrame.groupby()`, `Series.groupby()`, etc.

### 3.7.1 Indexing, iteration

---

`GroupBy.get_group(name)`

Construct DataFrame from group with provided name.

---

#### **databricks.koalas.groupby.GroupBy.get\_group**

`GroupBy.get_group(name)` → Union[databricks.koalas.frame.DataFrame,  
databricks.koalas.series.Series]  
Construct DataFrame from group with provided name.

##### **Parameters**

**name** [object] The name of the group to get as a DataFrame.

##### **Returns**

**group** [same type as obj]

## Examples

```
>>> kdf = ks.DataFrame([('falcon', 'bird', 389.0),
...                     ('parrot', 'bird', 24.0),
...                     ('lion', 'mammal', 80.5),
...                     ('monkey', 'mammal', np.nan)],
...                     columns=['name', 'class', 'max_speed'],
...                     index=[0, 2, 3, 1])
>>> kdf
   name  class  max_speed
0  falcon   bird     389.0
2  parrot   bird      24.0
3    lion  mammal     80.5
1  monkey  mammal      NaN
```

```
>>> kdf.groupby("class").get_group("bird").sort_index()
   name  class  max_speed
0  falcon   bird     389.0
2  parrot   bird      24.0
```

```
>>> kdf.groupby("class").get_group("mammal").sort_index()
   name  class  max_speed
1  monkey  mammal      NaN
3    lion  mammal     80.5
```

## 3.7.2 Function application

<code>GroupBy.apply(func, *args, **kwargs)</code>	Apply function <i>func</i> group-wise and combine the results together.
<code>GroupBy.transform(func, *args, **kwargs)</code>	Apply function column-by-column to the GroupBy object.

### databricks.koalas.groupby.GroupBy.apply

`GroupBy.apply(func, *args, **kwargs)` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]

Apply function *func* group-wise and combine the results together.

The function passed to *apply* must take a DataFrame as its first argument and return a DataFrame. *apply* will then take care of combining the results back together into a single dataframe. *apply* is therefore a highly flexible grouping method.

While *apply* is a very flexible method, its downside is that using it can be quite a bit slower than using more specific methods like *agg* or *transform*. Koalas offers a wide range of method that will be much faster than using *apply* for their specific purposes, so try to use them before reaching for *apply*.

**Note:** this API executes the function once to infer the type which is potentially expensive, for instance, when the dataset is created after aggregations or sorting.

To avoid this, specify return type in *func*, for instance, as below:

```
>>> def pandas_div(x) -> ks.DataFrame[float, float]:
...     return x[['B', 'C']] / x[['B', 'C']]
```



If the return type is specified, the output column names become *c0*, *c1*, *c2* ... *cn*. These names are positionally mapped to the returned DataFrame in *func*.

To specify the column names, you can assign them in a pandas friendly style as below:

```
>>> def pandas_div(x) -> ks.DataFrame["a": float, "b": float]:
...     return x[['B', 'C']] / x[['B', 'C']]

>>> pdf = pd.DataFrame({'B': [1.], 'C': [3.]})
>>> def plus_one(x) -> ks.DataFrame[zip(pdf.columns, pdf.dtypes)]:
...     return x[['B', 'C']] / x[['B', 'C']]
```

When the given function has the return type annotated, the original index of the GroupBy object will be lost and a default index will be attached to the result. Please be careful about configuring the default index. See also [Default Index Type](#).

**Note:** the dataframe within *func* is actually a pandas dataframe. Therefore, any pandas APIs within this function is allowed.

### Parameters

**func** [callable] A callable that takes a DataFrame as its first argument, and returns a dataframe.

**\*args** Positional arguments to pass to func.

**\*\*kwargs** Keyword arguments to pass to func.

### Returns

**applied** [DataFrame or Series]

See also:

**aggregate** Apply aggregate function to the GroupBy object.

**DataFrame.apply** Apply a function to a DataFrame.

**Series.apply** Apply a function to a Series.

### Examples

```
>>> df = ks.DataFrame({'A': 'a a b'.split(),
...                   'B': [1, 2, 3],
...                   'C': [4, 6, 5]}, columns=['A', 'B', 'C'])
>>> g = df.groupby('A')
```

Notice that *g* has two groups, a and b. Calling *apply* in various ways, we can get different grouping results:

Below the functions passed to *apply* takes a DataFrame as its argument and returns a DataFrame. *apply* combines the result for each group together into a new DataFrame:

```
>>> def plus_min(x):
...     return x + x.min()
>>> g.apply(plus_min).sort_index()
A  B  C
```

(continues on next page)

(continued from previous page)

```
0  aa  2   8
1  aa  3  10
2  bb  6  10
```

```
>>> g.apply(sum).sort_index()
      A  B   C
A
a  aa  3  10
b   b  3   5
```

```
>>> g.apply(len).sort_index()
A
a     2
b     1
dtype: int64
```

You can specify the type hint and prevent schema inference for better performance.

```
>>> def pandas_div(x) -> ks.DataFrame[float, float]:
...     return x[['B', 'C']] / x[['B', 'C']]
>>> g.apply(pandas_div).sort_index()
      c0   c1
0  1.0  1.0
1  1.0  1.0
2  1.0  1.0
```

In case of Series, it works as below.

```
>>> def plus_max(x) -> ks.Series[np.int]:
...     return x + x.max()
>>> df.B.groupby(df.A).apply(plus_max).sort_index()
0     6
1     3
2     4
Name: B, dtype: int64
```

```
>>> def plus_min(x):
...     return x + x.min()
>>> df.B.groupby(df.A).apply(plus_min).sort_index()
0     2
1     3
2     6
Name: B, dtype: int64
```

You can also return a scalar value as a aggregated value of the group:

```
>>> def plus_length(x) -> np.int:
...     return len(x)
>>> df.B.groupby(df.A).apply(plus_length).sort_index()
0     1
1     2
Name: B, dtype: int64
```

The extra arguments to the function can be passed as below.

```
>>> def calculation(x, y, z) -> np.int:
...     return len(x) + y * z
>>> df.B.groupby(df.A).apply(calculation, 5, z=10).sort_index()
0      51
1      52
Name: B, dtype: int64
```

### databricks.koalas.groupby.GroupBy.transform

`GroupBy.transform(func, *args, **kwargs)` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]

Apply function column-by-column to the GroupBy object.

The function passed to *transform* must take a Series as its first argument and return a Series. The given function is executed for each series in each grouped data.

While *transform* is a very flexible method, its downside is that using it can be quite a bit slower than using more specific methods like *agg* or *transform*. Koalas offers a wide range of method that will be much faster than using *transform* for their specific purposes, so try to use them before reaching for *transform*.

---

#### Note:

**this API executes the function once to infer the type which is** potentially expensive, for instance, when the dataset is created after aggregations or sorting.

To avoid this, specify return type in `func`, for instance, as below:

```
>>> def convert_to_string(x) -> ks.Series[str]:
...     return x.apply("a string {}".format)
```

When the given function has the return type annotated, the original index of the GroupBy object will be lost and a default index will be attached to the result. Please be careful about configuring the default index. See also [Default Index Type](#).

---

**Note:** the series within `func` is actually a pandas series. Therefore, any pandas APIs within this function is allowed.

---

#### Parameters

**func** [callable] A callable that takes a Series as its first argument, and returns a Series.

**\*args** Positional arguments to pass to func.

**\*\*kwargs** Keyword arguments to pass to func.

#### Returns

**applied** [DataFrame]

See also:

**aggregate** Apply aggregate function to the GroupBy object.

**Series.apply** Apply a function to a Series.

## Examples

```
>>> df = ks.DataFrame({'A': [0, 0, 1],
...                     'B': [1, 2, 3],
...                     'C': [4, 6, 5]}, columns=['A', 'B', 'C'])
```

```
>>> g = df.groupby('A')
```

Notice that `g` has two groups, 0 and 1. Calling *transform* in various ways, we can get different grouping results: Below the functions passed to *transform* takes a Series as its argument and returns a Series. *transform* applies the function on each series in each grouped data, and combine them into a new DataFrame:

```
>>> def convert_to_string(x) -> ks.Series[str]:
...     return x.apply("a string {}".format)
>>> g.transform(convert_to_string)
      B      C
0  a string 1  a string 4
1  a string 2  a string 6
2  a string 3  a string 5
```

```
>>> def plus_max(x) -> ks.Series[np.int]:
...     return x + x.max()
>>> g.transform(plus_max)
      B      C
0     3     10
1     4     12
2     6     10
```

You can omit the type hint and let Koalas infer its type.

```
>>> def plus_min(x):
...     return x + x.min()
>>> g.transform(plus_min)
      B      C
0     2      8
1     3     10
2     6     10
```

In case of Series, it works as below.

```
>>> df.B.groupby(df.A).transform(plus_max)
0      3
1      4
2      6
Name: B, dtype: int64
```

```
>>> (df * -1).B.groupby(df.A).transform(abs)
0      1
1      2
2      3
Name: B, dtype: int64
```

You can also specify extra arguments to pass to the function.

```
>>> def calculation(x, y, z) -> ks.Series[np.int]:
...     return x + x.min() + y + z
```

(continues on next page)

(continued from previous page)

```
>>> g.transform(calculation, 5, z=20)
   B  C
0  27 33
1  28 35
2  31 35
```

The following methods are available only for *DataFrameGroupBy* objects.

<code>DataFrameGroupBy.agg([func_or_funcs])</code>	Aggregate using one or more operations over the specified axis.
<code>DataFrameGroupBy.aggregate([func_or_funcs])</code>	Aggregate using one or more operations over the specified axis.

### `databricks.koalas.groupby.DataFrameGroupBy.agg`

`DataFrameGroupBy.agg(func_or_funcs=None, *args, **kwargs) → databricks.koalas.frame.DataFrame`  
 Aggregate using one or more operations over the specified axis.

#### Parameters

**func\_or\_funcs** [dict, str or list] a dict mapping from column name (string) to aggregate functions (string or list of strings).

#### Returns

**Series or DataFrame** The return can be:

- Series : when `DataFrame.agg` is called with a single function
- DataFrame : when `DataFrame.agg` is called with several functions

Return Series or DataFrame.

See also:

`databricks.koalas.Series.groupby`

`databricks.koalas.DataFrame.groupby`

#### Notes

`agg` is an alias for `aggregate`. Use the alias.

#### Examples

```
>>> df = ks.DataFrame({'A': [1, 1, 2, 2],
...                    'B': [1, 2, 3, 4],
...                    'C': [0.362, 0.227, 1.267, -0.562]}),
...                    columns=['A', 'B', 'C'])
```

```
>>> df
   A  B    C
0  1  1  0.362
1  1  2  0.227
```

(continues on next page)

(continued from previous page)

```
2  2  3  1.267
3  2  4 -0.562
```

**Different aggregations per column**

```
>>> aggregated = df.groupby('A').agg({'B': 'min', 'C': 'sum'})
>>> aggregated[['B', 'C']].sort_index()
      B      C
A
1  1  0.589
2  3  0.705
```

```
>>> aggregated = df.groupby('A').agg({'B': ['min', 'max']})
>>> aggregated.sort_index()
      B
      min  max
A
1      1     2
2      3     4
```

```
>>> aggregated = df.groupby('A').agg('min')
>>> aggregated.sort_index()
      B      C
A
1      1  0.227
2      3 -0.562
```

```
>>> aggregated = df.groupby('A').agg(['min', 'max'])
>>> aggregated.sort_index()
      B      C
      min  max  min  max
A
1      1     2  0.227  0.362
2      3     4 -0.562  1.267
```

To control the output names with different aggregations per column, Koalas also supports ‘named aggregation’ or nested renaming in `.agg`. It can also be used when applying multiple aggregation functions to specific columns.

```
>>> aggregated = df.groupby('A').agg(b_max=ks.NamedAgg(column='B', aggfunc='max'))
>>> aggregated.sort_index()
      b_max
A
1          2
2          4
```

```
>>> aggregated = df.groupby('A').agg(b_max=('B', 'max'), b_min=('B', 'min'))
>>> aggregated.sort_index()
      b_max  b_min
A
1          2      1
2          4      3
```

```
>>> aggregated = df.groupby('A').agg(b_max=('B', 'max'), c_min=('C', 'min'))
>>> aggregated.sort_index()
```

(continues on next page)

(continued from previous page)

	b_max	c_min
A		
1	2	0.227
2	4	-0.562

**databricks.koalas.groupby.DataFrameGroupBy.aggregate**

DataFrameGroupBy.**aggregate** (*func\_or\_funcs=None*, *\*args*, *\*\*kwargs*) →

databricks.koalas.frame.DataFrame  
Aggregate using one or more operations over the specified axis.

**Parameters**

**func\_or\_funcs** [dict, str or list] a dict mapping from column name (string) to aggregate functions (string or list of strings).

**Returns**

**Series or DataFrame** The return can be:

- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return Series or DataFrame.

See also:

[\*databricks.koalas.Series.groupby\*](#)

[\*databricks.koalas.DataFrame.groupby\*](#)

**Notes**

*agg* is an alias for *aggregate*. Use the alias.

**Examples**

```
>>> df = ks.DataFrame({'A': [1, 1, 2, 2],
...                     'B': [1, 2, 3, 4],
...                     'C': [0.362, 0.227, 1.267, -0.562]}),
...                   columns=['A', 'B', 'C'])
```

```
>>> df
   A  B    C
0  1  1  0.362
1  1  2  0.227
2  2  3  1.267
3  2  4 -0.562
```

Different aggregations per column

```
>>> aggregated = df.groupby('A').agg({'B': 'min', 'C': 'sum'})
>>> aggregated[['B', 'C']].sort_index()
   B    C
```

(continues on next page)

(continued from previous page)

```
A
1  1  0.589
2  3  0.705
```

```
>>> aggregated = df.groupby('A').agg({'B': ['min', 'max']})
>>> aggregated.sort_index()
      B
      min max
A
1      1   2
2      3   4
```

```
>>> aggregated = df.groupby('A').agg('min')
>>> aggregated.sort_index()
      B      C
A
1      1  0.227
2      3 -0.562
```

```
>>> aggregated = df.groupby('A').agg(['min', 'max'])
>>> aggregated.sort_index()
      B      C
      min max  min  max
A
1      1   2  0.227  0.362
2      3   4 -0.562  1.267
```

To control the output names with different aggregations per column, Koalas also supports ‘named aggregation’ or nested renaming in `.agg`. It can also be used when applying multiple aggregation functions to specific columns.

```
>>> aggregated = df.groupby('A').agg(b_max=ks.NamedAgg(column='B', aggfunc='max'))
>>> aggregated.sort_index()
      b_max
A
1          2
2          4
```

```
>>> aggregated = df.groupby('A').agg(b_max=('B', 'max'), b_min=('B', 'min'))
>>> aggregated.sort_index()
      b_max  b_min
A
1          2      1
2          4      3
```

```
>>> aggregated = df.groupby('A').agg(b_max=('B', 'max'), c_min=('C', 'min'))
>>> aggregated.sort_index()
      b_max  c_min
A
1          2  0.227
2          4 -0.562
```



### 3.7.3 Computations / Descriptive Stats

<code>GroupBy.all()</code>	Returns True if all values in the group are truthful, else False.
<code>GroupBy.any()</code>	Returns True if any value in the group is truthful, else False.
<code>GroupBy.count()</code>	Compute count of group, excluding missing values.
<code>GroupBy.cumcount([ascending])</code>	Number each item in each group from 0 to the length of that group - 1.
<code>GroupBy.cummax()</code>	Cumulative max for each group.
<code>GroupBy.cummin()</code>	Cumulative min for each group.
<code>GroupBy.cumprod()</code>	Cumulative product for each group.
<code>GroupBy.cumsum()</code>	Cumulative sum for each group.
<code>GroupBy.filter(func)</code>	Return a copy of a DataFrame excluding elements from groups that do not satisfy the boolean criterion specified by func.
<code>GroupBy.first()</code>	Compute first of group values.
<code>GroupBy.last()</code>	Compute last of group values.
<code>GroupBy.max()</code>	Compute max of group values.
<code>GroupBy.mean()</code>	Compute mean of groups, excluding missing values.
<code>GroupBy.median([numeric_only, accuracy])</code>	Compute median of groups, excluding missing values.
<code>GroupBy.min()</code>	Compute min of group values.
<code>GroupBy.rank([method, ascending])</code>	Provide the rank of values within each group.
<code>GroupBy.std([ddof])</code>	Compute standard deviation of groups, excluding missing values.
<code>GroupBy.sum()</code>	Compute sum of group values
<code>GroupBy.var([ddof])</code>	Compute variance of groups, excluding missing values.
<code>GroupBy.nunique([dropna])</code>	Return DataFrame with number of distinct observations per group for each column.
<code>GroupBy.size()</code>	Compute group sizes.
<code>GroupBy.diff([periods])</code>	First discrete difference of element.
<code>GroupBy.idxmax([skipna])</code>	Return index of first occurrence of maximum over requested axis in group.
<code>GroupBy.idxmin([skipna])</code>	Return index of first occurrence of minimum over requested axis in group.
<code>GroupBy.fillna([value, method, axis, ...])</code>	Fill NA/NaN values in group.
<code>GroupBy.bfill([limit])</code>	Synonym for <code>DataFrame.fillna()</code> with <code>method='bfill'</code> .
<code>GroupBy.ffill([limit])</code>	Synonym for <code>DataFrame.fillna()</code> with <code>method='ffill'</code> .
<code>GroupBy.head([n])</code>	Return first n rows of each group.
<code>GroupBy.backfill([limit])</code>	Synonym for <code>DataFrame.fillna()</code> with <code>method='bfill'</code> .
<code>GroupBy.shift([periods, fill_value])</code>	Shift each group by periods observations.
<code>GroupBy.tail([n])</code>	Return last n rows of each group.

**databricks.koalas.groupby.GroupBy.all**

`GroupBy.all()` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
Returns True if all values in the group are truthful, else False.

See also:

*databricks.koalas.Series.groupby*

*databricks.koalas.DataFrame.groupby*

**Examples**

```
>>> df = ks.DataFrame({'A': [1, 1, 2, 2, 3, 3, 4, 4, 5, 5],
...                   'B': [True, True, True, False, False,
...                   False, None, True, None, False]},
...                   columns=['A', 'B'])
>>> df
   A  B
0  1  True
1  1  True
2  2  True
3  2 False
4  3 False
5  3 False
6  4  None
7  4  True
8  5  None
9  5 False
```

```
>>> df.groupby('A').all().sort_index()
   B
A
1  True
2 False
3 False
4  True
5 False
```

**databricks.koalas.groupby.GroupBy.any**

`GroupBy.any()` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
Returns True if any value in the group is truthful, else False.

See also:

*databricks.koalas.Series.groupby*

*databricks.koalas.DataFrame.groupby*

## Examples

```
>>> df = ks.DataFrame({'A': [1, 1, 2, 2, 3, 3, 4, 4, 5, 5],
...                    'B': [True, True, True, False, False,
...                          False, None, True, None, False]},
...                   columns=['A', 'B'])
>>> df
   A      B
0  1   True
1  1   True
2  2   True
3  2  False
4  3  False
5  3  False
6  4   None
7  4   True
8  5   None
9  5  False
```

```
>>> df.groupby('A').any().sort_index()
      B
A
1   True
2   True
3  False
4   True
5  False
```

### `databricks.koalas.groupby.GroupBy.count`

`GroupBy.count()` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
 Compute count of group, excluding missing values.

See also:

`databricks.koalas.Series.groupby`

`databricks.koalas.DataFrame.groupby`

## Examples

```
>>> df = ks.DataFrame({'A': [1, 1, 2, 1, 2],
...                    'B': [np.nan, 2, 3, 4, 5],
...                    'C': [1, 2, 1, 1, 2]}, columns=['A', 'B', 'C'])
>>> df.groupby('A').count().sort_index()
      B  C
A
1  2  3
2  2  2
```

**databricks.koalas.groupby.GroupBy.cumcount**

`GroupBy.cumcount (ascending=True)` → `databricks.koalas.series.Series`  
Number each item in each group from 0 to the length of that group - 1.

Essentially this is equivalent to

```
self.apply(lambda x: pd.Series(np.arange(len(x)), x.index))
```

**Parameters**

**ascending** [bool, default True] If False, number in reverse, from length of group - 1 to 0.

**Returns**

**Series** Sequence number of each element within each group.

**Examples**

```
>>> df = ks.DataFrame(['a'], ['a'], ['a'], ['b'], ['b'], ['a']],
...                    columns=['A'])
>>> df
   A
0  a
1  a
2  a
3  b
4  b
5  a
>>> df.groupby('A').cumcount().sort_index()
0    0
1    1
2    2
3    0
4    1
5    3
dtype: int64
>>> df.groupby('A').cumcount(ascending=False).sort_index()
0    3
1    2
2    1
3    1
4    0
5    0
dtype: int64
```

**databricks.koalas.groupby.GroupBy.cummax**

`GroupBy.cummax()` → `Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]`  
Cumulative max for each group.

**Returns**

**Series or DataFrame**

**See also:**

**Series.cummax****DataFrame.cummax**

## Examples

```
>>> df = ks.DataFrame(
...     [[1, None, 4], [1, 0.1, 3], [1, 20.0, 2], [4, 10.0, 1]],
...     columns=list('ABC'))
>>> df
```

	A	B	C
0	1	NaN	4
1	1	0.1	3
2	1	20.0	2
3	4	10.0	1

By default, iterates over rows and finds the sum in each column.

```
>>> df.groupby("A").cummax().sort_index()
```

	B	C
0	NaN	4
1	0.1	4
2	20.0	4
3	10.0	1

It works as below in Series.

```
>>> df.C.groupby(df.A).cummax().sort_index()
```

0	4
1	4
2	4
3	1

Name: C, dtype: int64

## databricks.koalas.groupby.GroupBy.cummin

`GroupBy.cummin()` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
Cumulative min for each group.

### Returns

**Series or DataFrame**

**See also:**

**Series.cummin**

**DataFrame.cummin**

## Examples

```
>>> df = ks.DataFrame(
...     [[1, None, 4], [1, 0.1, 3], [1, 20.0, 2], [4, 10.0, 1]],
...     columns=list('ABC'))
>>> df
   A      B  C
0  1   NaN  4
1  1   0.1  3
2  1  20.0  2
3  4  10.0  1
```

By default, iterates over rows and finds the sum in each column.

```
>>> df.groupby("A").cummin().sort_index()
      B  C
0   NaN  4
1   0.1  3
2   0.1  2
3  10.0  1
```

It works as below in Series.

```
>>> df.B.groupby(df.A).cummin().sort_index()
0      NaN
1     0.1
2     0.1
3    10.0
Name: B, dtype: float64
```

## databricks.koalas.groupby.GroupBy.cumprod

GroupBy.**cumprod**() → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
Cumulative product for each group.

### Returns

**Series or DataFrame**

See also:

**Series.cumprod**

**DataFrame.cumprod**

## Examples

```
>>> df = ks.DataFrame(
...     [[1, None, 4], [1, 0.1, 3], [1, 20.0, 2], [4, 10.0, 1]],
...     columns=list('ABC'))
>>> df
   A      B  C
0  1   NaN  4
1  1   0.1  3
2  1  20.0  2
3  4  10.0  1
```

By default, iterates over rows and finds the sum in each column.

```
>>> df.groupby("A").cumprod().sort_index()
      B    C
0  NaN    4
1   0.1   12
2   2.0   24
3  10.0    1
```

It works as below in Series.

```
>>> df.B.groupby(df.A).cumprod().sort_index()
0      NaN
1     0.1
2     2.0
3    10.0
Name: B, dtype: float64
```

### **databricks.koalas.groupby.GroupBy.cumsum**

`GroupBy.cumsum()` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
Cumulative sum for each group.

#### **Returns**

**Series or DataFrame**

See also:

**Series.cumsum**

**DataFrame.cumsum**

### **Examples**

```
>>> df = ks.DataFrame(
...     [[1, None, 4], [1, 0.1, 3], [1, 20.0, 2], [4, 10.0, 1]],
...     columns=list('ABC'))
>>> df
      A    B    C
0   1  NaN    4
1   1   0.1    3
2   1  20.0    2
3   4  10.0    1
```

By default, iterates over rows and finds the sum in each column.

```
>>> df.groupby("A").cumsum().sort_index()
      B    C
0  NaN    4
1   0.1    7
2  20.1    9
3  10.0    1
```

It works as below in Series.

```
>>> df.B.groupby(df.A).cumsum().sort_index()
0      NaN
1      0.1
2     20.1
3     10.0
Name: B, dtype: float64
```

### **databricks.koalas.groupby.GroupBy.filter**

`GroupBy.filter(func) → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]`

Return a copy of a DataFrame excluding elements from groups that do not satisfy the boolean criterion specified by func.

#### **Parameters**

**f** [function] Function to apply to each subframe. Should return True or False.

**dropna** [Drop groups that do not pass the filter. True by default;] if False, groups that evaluate False are filled with NaNs.

#### **Returns**

**filtered** [DataFrame or Series]

### **Notes**

Each subframe is endowed the attribute 'name' in case you need to know which group you are working on.

### **Examples**

```
>>> df = ks.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
...                          'foo', 'bar'],
...                   'B' : [1, 2, 3, 4, 5, 6],
...                   'C' : [2.0, 5., 8., 1., 2., 9.]}, columns=['A', 'B', 'C'])
>>> grouped = df.groupby('A')
>>> grouped.filter(lambda x: x['B'].mean() > 3.)
   A  B    C
1  bar  2  5.0
3  bar  4  1.0
5  bar  6  9.0
```

```
>>> df.B.groupby(df.A).filter(lambda x: x.mean() > 3.)
1      2
3      4
5      6
Name: B, dtype: int64
```



### **databricks.koalas.groupby.GroupBy.first**

`GroupBy.first()` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
Compute first of group values.

See also:

*databricks.koalas.Series.groupby*

*databricks.koalas.DataFrame.groupby*

### **databricks.koalas.groupby.GroupBy.last**

`GroupBy.last()` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
Compute last of group values.

See also:

*databricks.koalas.Series.groupby*

*databricks.koalas.DataFrame.groupby*

### **databricks.koalas.groupby.GroupBy.max**

`GroupBy.max()` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
Compute max of group values.

See also:

*databricks.koalas.Series.groupby*

*databricks.koalas.DataFrame.groupby*

### **databricks.koalas.groupby.GroupBy.mean**

`GroupBy.mean()` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
Compute mean of groups, excluding missing values.

**Returns**

**koalas.Series or koalas.DataFrame**

See also:

*databricks.koalas.Series.groupby*

*databricks.koalas.DataFrame.groupby*

## Examples

```
>>> df = ks.DataFrame({'A': [1, 1, 2, 1, 2],
...                    'B': [np.nan, 2, 3, 4, 5],
...                    'C': [1, 2, 1, 1, 2]}, columns=['A', 'B', 'C'])
```

Groupby one column and return the mean of the remaining columns in each group.

```
>>> df.groupby('A').mean().sort_index()
      B      C
A
1  3.0  1.333333
2  4.0  1.500000
```

## databricks.koalas.groupby.GroupBy.median

GroupBy.**median** (*numeric\_only=True*, *accuracy=10000*) → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]

Compute median of groups, excluding missing values.

For multiple groupings, the result index will be a MultiIndex

---

**Note:** Unlike pandas', the median in Koalas is an approximated median based upon approximate percentile computation because computing median across a large dataset is extremely expensive.

---

### Parameters

**numeric\_only** [bool, default True] Include only float, int, boolean columns. False is not supported. This parameter is mainly for pandas compatibility.

### Returns

**Series or DataFrame** Median of values within each group.

## Examples

```
>>> kdf = ks.DataFrame({'a': [1., 1., 1., 1., 2., 2., 2., 3., 3., 3.],
...                    'b': [2., 3., 1., 4., 6., 9., 8., 10., 7., 5.],
...                    'c': [3., 5., 2., 5., 1., 2., 6., 4., 3., 6.]},
...                    columns=['a', 'b', 'c'],
...                    index=[7, 2, 4, 1, 3, 4, 9, 10, 5, 6])
>>> kdf
      a      b      c
7  1.0  2.0  3.0
2  1.0  3.0  5.0
4  1.0  1.0  2.0
1  1.0  4.0  5.0
3  2.0  6.0  1.0
4  2.0  9.0  2.0
9  2.0  8.0  6.0
10 3.0 10.0  4.0
5  3.0  7.0  3.0
6  3.0  5.0  6.0
```

### DataFrameGroupBy

```
>>> kdf.groupby('a').median().sort_index()
      b      c
a
1.0  2.0  3.0
2.0  8.0  2.0
3.0  7.0  4.0
```

### SeriesGroupBy

```
>>> kdf.groupby('a')['b'].median().sort_index()
a
1.0    2.0
2.0    8.0
3.0    7.0
Name: b, dtype: float64
```

## databricks.koalas.groupby.GroupBy.min

`GroupBy.min()` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
 Compute min of group values.

**See also:**

*databricks.koalas.Series.groupby*

*databricks.koalas.DataFrame.groupby*

## databricks.koalas.groupby.GroupBy.rank

`GroupBy.rank(method='average', ascending=True)` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
 Provide the rank of values within each group.

### Parameters

**method** [{‘average’, ‘min’, ‘max’, ‘first’, ‘dense’}, default ‘average’]

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like ‘min’, but rank always increases by 1 between groups

**ascending** [boolean, default True] False for ranks by high (1) to low (N)

### Returns

**DataFrame with ranking of values within each group**

## Examples

```
>>> df = ks.DataFrame({
...     'a': [1, 1, 1, 2, 2, 2, 3, 3, 3],
...     'b': [1, 2, 2, 2, 3, 3, 3, 4, 4]}, columns=['a', 'b'])
>>> df
   a  b
0  1  1
1  1  2
2  1  2
3  2  2
4  2  3
5  2  3
6  3  3
7  3  4
8  3  4
```

```
>>> df.groupby("a").rank().sort_index()
   b
0  1.0
1  2.5
2  2.5
3  1.0
4  2.5
5  2.5
6  1.0
7  2.5
8  2.5
```

```
>>> df.b.groupby(df.a).rank(method='max').sort_index()
0    1.0
1    3.0
2    3.0
3    1.0
4    3.0
5    3.0
6    1.0
7    3.0
8    3.0
Name: b, dtype: float64
```

## `databricks.koalas.groupby.GroupBy.std`

`GroupBy.std(ddof: int = 1) → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]`  
 Compute standard deviation of groups, excluding missing values.

### Parameters

**ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

See also:

`databricks.koalas.Series.groupby`

`databricks.koalas.DataFrame.groupby`

**databricks.koalas.groupby.GroupBy.sum**

`GroupBy.sum()` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
 Compute sum of group values

See also:

*databricks.koalas.Series.groupby*

*databricks.koalas.DataFrame.groupby*

**databricks.koalas.groupby.GroupBy.var**

`GroupBy.var(ddof: int = 1)` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
 Compute variance of groups, excluding missing values.

**Parameters**

**ddof** [int, default 1] Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

See also:

*databricks.koalas.Series.groupby*

*databricks.koalas.DataFrame.groupby*

**databricks.koalas.groupby.GroupBy.nunique**

`GroupBy.nunique(dropna=True)` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
 Return DataFrame with number of distinct observations per group for each column.

**Parameters**

**dropna** [boolean, default True] Don't include NaN in the counts.

**Returns**

**nunique** [DataFrame or Series]

**Examples**

```
>>> df = ks.DataFrame({'id': ['spam', 'egg', 'egg', 'spam',
...                           'ham', 'ham'],
...                    'value1': [1, 5, 5, 2, 5, 5],
...                    'value2': list('abbaxy')}, columns=['id', 'value1', 'value2'])
>>> df
   id  value1 value2
0 spam      1      a
1 egg      5      b
2 egg      5      b
3 spam      2      a
4 ham      5      x
5 ham      5      y
```

```
>>> df.groupby('id').unique().sort_index()
      value1  value2
id
egg         1       1
ham         1       2
spam        2       1
```

```
>>> df.groupby('id')['value1'].unique().sort_index()
id
egg     1
ham     1
spam    2
Name: value1, dtype: int64
```

### **databricks.koalas.groupby.GroupBy.size**

GroupBy.**size**() → databricks.koalas.series.Series

Compute group sizes.

**See also:**

*databricks.koalas.Series.groupby*

*databricks.koalas.DataFrame.groupby*

### **Examples**

```
>>> df = ks.DataFrame({'A': [1, 2, 2, 3, 3, 3],
...                    'B': [1, 1, 2, 3, 3, 3]},
...                    columns=['A', 'B'])
>>> df
   A  B
0  1  1
1  2  1
2  2  2
3  3  3
4  3  3
5  3  3
```

```
>>> df.groupby('A').size().sort_index()
A
1     1
2     2
3     3
dtype: int64
```

```
>>> df.groupby(['A', 'B']).size().sort_index()
A  B
1  1     1
2  1     1
   2     1
3  3     3
dtype: int64
```

For Series,

```
>>> df.B.groupby(df.A).size().sort_index()
A
1      1
2      2
3      3
Name: B, dtype: int64
```

```
>>> df.groupby(df.A).B.size().sort_index()
A
1      1
2      2
3      3
Name: B, dtype: int64
```

### `databricks.koalas.groupby.GroupBy.diff`

`GroupBy.diff(periods=1)` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame group (default is the element in the same column of the previous row).

#### Parameters

**periods** [int, default 1] Periods to shift for calculating difference, accepts negative values.

#### Returns

**diffed** [DataFrame or Series]

See also:

`databricks.koalas.Series.groupby`

`databricks.koalas.DataFrame.groupby`

### Examples

```
>>> df = ks.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]}, columns=['a', 'b', 'c'])
>>> df
   a  b  c
0  1  1  1
1  2  1  4
2  3  2  9
3  4  3 16
4  5  5 25
5  6  8 36
```

```
>>> df.groupby(['b']).diff().sort_index()
   a  c
0 NaN NaN
1 1.0 3.0
```

(continues on next page)

(continued from previous page)

```

2  NaN  NaN
3  NaN  NaN
4  NaN  NaN
5  NaN  NaN

```

Difference with previous column in a group.

```

>>> df.groupby(['b'])['a'].diff().sort_index()
0      NaN
1      1.0
2      NaN
3      NaN
4      NaN
5      NaN
Name: a, dtype: float64

```

### `databricks.koalas.groupby.GroupBy.idxmax`

`GroupBy.idxmax(skipna=True)` → `Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]`

Return index of first occurrence of maximum over requested axis in group. NA/null values are excluded.

#### Parameters

**skipna** [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

See also:

`Series.idxmax`

`DataFrame.idxmax`

`databricks.koalas.Series.groupby`

`databricks.koalas.DataFrame.groupby`

#### Examples

```

>>> df = ks.DataFrame({'a': [1, 1, 2, 2, 3],
...                    'b': [1, 2, 3, 4, 5],
...                    'c': [5, 4, 3, 2, 1]}, columns=['a', 'b', 'c'])

```

```

>>> df.groupby(['a'])['b'].idxmax().sort_index()
a
1  1
2  3
3  4
Name: b, dtype: int64

```

```

>>> df.groupby(['a']).idxmax().sort_index()
   b  c
a
1  1  0

```

(continues on next page)



(continued from previous page)

```
2 3 2
3 4 4
```

**databricks.koalas.groupby.GroupBy.idxmin**

`GroupBy.idxmin(skipna=True)` → `Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]`  
 Return index of first occurrence of minimum over requested axis in group. NA/null values are excluded.

**Parameters**

**skipna** [boolean, default True] Exclude NA/null values. If an entire row/column is NA, the result will be NA.

See also:

**Series.idxmin**

**DataFrame.idxmin**

*databricks.koalas.Series.groupby*

*databricks.koalas.DataFrame.groupby*

**Examples**

```
>>> df = ks.DataFrame({'a': [1, 1, 2, 2, 3],
...                    'b': [1, 2, 3, 4, 5],
...                    'c': [5, 4, 3, 2, 1]}, columns=['a', 'b', 'c'])
```

```
>>> df.groupby(['a'])['b'].idxmin().sort_index()
a
1    0
2    2
3    4
Name: b, dtype: int64
```

```
>>> df.groupby(['a']).idxmin().sort_index()
   b  c
a
1  0  1
2  2  3
3  4  4
```

**databricks.koalas.groupby.GroupBy.fillna**

`GroupBy.fillna` (*value=None, method=None, axis=None, inplace=False, limit=None*) →  
 Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
 Fill NA/NaN values in group.

**Parameters**

**value** [scalar, dict, Series] Value to use to fill holes. alternately a dict/Series of values specifying which value to use for each column. DataFrame is not supported.

**method** [{‘backfill’, ‘bfill’, ‘pad’, ‘ffill’, None}, default None] Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

**axis** [{0 or index}] 1 and *columns* are not supported.

**inplace** [boolean, default False] Fill in place (do not create a new object)

**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

**Returns**

**DataFrame** DataFrame with NA entries filled.

**Examples**

```
>>> df = ks.DataFrame({
...     'A': [1, 1, 2, 2],
...     'B': [2, 4, None, 3],
...     'C': [None, None, None, 1],
...     'D': [0, 1, 5, 4]
... },
...     columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C  D
0  1  2.0 NaN 0
1  1  4.0 NaN 1
2  2  NaN NaN 5
3  2  3.0 1.0 4
```

We can also propagate non-null values forward or backward in group.

```
>>> df.groupby(['A'])['B'].fillna(method='ffill').sort_index()
0    2.0
1    4.0
2    NaN
3    3.0
Name: B, dtype: float64
```

```
>>> df.groupby(['A']).fillna(method='bfill').sort_index()
   B   C  D
0  2.0 NaN 0
1  4.0 NaN 1
```

(continues on next page)

(continued from previous page)

2	3.0	1.0	5
3	3.0	1.0	4

**databricks.koalas.groupby.GroupBy.bfill**

GroupBy.**bfill** (*limit=None*) → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
 Synonym for *DataFrame.fillna()* with *method='bfill'*.

**Parameters**

**axis** [{0 or *index*}] 1 and *columns* are not supported.

**inplace** [boolean, default False] Fill in place (do not create a new object)

**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

**Returns**

**DataFrame** DataFrame with NA entries filled.

**Examples**

```
>>> df = ks.DataFrame({
...     'A': [1, 1, 2, 2],
...     'B': [2, 4, None, 3],
...     'C': [None, None, None, 1],
...     'D': [0, 1, 5, 4]
... },
...     columns=['A', 'B', 'C', 'D'])
>>> df
   A  B    C  D
0  1  2.0 NaN  0
1  1  4.0 NaN  1
2  2  NaN NaN  5
3  2  3.0 1.0  4
```

Propagate non-null values backward.

```
>>> df.groupby(['A']).bfill().sort_index()
   B    C  D
0  2.0 NaN  0
1  4.0 NaN  1
2  3.0 1.0  5
3  3.0 1.0  4
```

**databricks.koalas.groupby.GroupBy.ffill**

`GroupBy.ffill (limit=None)` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
 Synonym for `DataFrame.fillna()` with `method='ffill'`.

**Parameters**

**axis** [[0 or *index*]] 1 and *columns* are not supported.

**inplace** [boolean, default False] Fill in place (do not create a new object)

**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

**Returns**

**DataFrame** DataFrame with NA entries filled.

**Examples**

```
>>> df = ks.DataFrame({
...     'A': [1, 1, 2, 2],
...     'B': [2, 4, None, 3],
...     'C': [None, None, None, 1],
...     'D': [0, 1, 5, 4]
... },
...     columns=['A', 'B', 'C', 'D'])
>>> df
   A  B    C  D
0  1  2.0 NaN  0
1  1  4.0 NaN  1
2  2  NaN NaN  5
3  2  3.0 1.0  4
```

Propagate non-null values forward.

```
>>> df.groupby(['A']).ffill().sort_index()
   B    C  D
0  2.0 NaN  0
1  4.0 NaN  1
2  NaN NaN  5
3  3.0 1.0  4
```

**databricks.koalas.groupby.GroupBy.head**

`GroupBy.head (n=5)` → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
 Return first n rows of each group.

**Returns**

**DataFrame or Series**

## Examples

```
>>> df = ks.DataFrame({'a': [1, 1, 1, 1, 2, 2, 2, 3, 3, 3],
...                    'b': [2, 3, 1, 4, 6, 9, 8, 10, 7, 5],
...                    'c': [3, 5, 2, 5, 1, 2, 6, 4, 3, 6]},
...                    columns=['a', 'b', 'c'],
...                    index=[7, 2, 4, 1, 3, 4, 9, 10, 5, 6])
>>> df
   a  b  c
7  1  2  3
2  1  3  5
4  1  1  2
1  1  4  5
3  2  6  1
4  2  9  2
9  2  8  6
10 3 10  4
5  3  7  3
6  3  5  6
```

```
>>> df.groupby('a').head(2).sort_index()
   a  b  c
2  1  3  5
3  2  6  1
4  2  9  2
5  3  7  3
7  1  2  3
10 3 10  4
```

```
>>> df.groupby('a')['b'].head(2).sort_index()
2      3
3      6
4      9
5      7
7      2
10     10
Name: b, dtype: int64
```

### databricks.koalas.groupby.GroupBy.backfill

GroupBy.**backfill** (*limit=None*) → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
 Synonym for *DataFrame.fillna()* with *method='bfill'*.

#### Parameters

**axis** [{0 or *index*}] 1 and *columns* are not supported.

**inplace** [boolean, default False] Fill in place (do not create a new object)

**limit** [int, default None] If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None

#### Returns

**DataFrame** DataFrame with NA entries filled.

### Examples

```
>>> df = ks.DataFrame({
...     'A': [1, 1, 2, 2],
...     'B': [2, 4, None, 3],
...     'C': [None, None, None, 1],
...     'D': [0, 1, 5, 4]
... },
...     columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C  D
0  1  2.0 NaN 0
1  1  4.0 NaN 1
2  2  NaN NaN 5
3  2  3.0 1.0 4
```

Propagate non-null values backward.

```
>>> df.groupby(['A']).bfill().sort_index()
   B   C  D
0  2.0 NaN 0
1  4.0 NaN 1
2  3.0 1.0 5
3  3.0 1.0 4
```

### **databricks.koalas.groupby.GroupBy.shift**

`GroupBy.shift` (*periods=1*, *fill\_value=None*) → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]  
Shift each group by periods observations.

#### Parameters

**periods** [integer, default 1] number of periods to shift

**fill\_value** [optional]

#### Returns

**Series or DataFrame** Object shifted within each group.

### Examples

```
>>> df = ks.DataFrame({
...     'a': [1, 1, 1, 2, 2, 2, 3, 3, 3],
...     'b': [1, 2, 2, 2, 3, 3, 3, 4, 4]}, columns=['a', 'b'])
>>> df
   a  b
0  1  1
1  1  2
2  1  2
3  2  2
4  2  3
```

(continues on next page)

(continued from previous page)

```

5  2  3
6  3  3
7  3  4
8  3  4

```

```

>>> df.groupby('a').shift().sort_index()
      b
0  NaN
1  1.0
2  2.0
3  NaN
4  2.0
5  3.0
6  NaN
7  3.0
8  4.0

```

```

>>> df.groupby('a').shift(periods=-1, fill_value=0).sort_index()
      b
0  2
1  2
2  0
3  3
4  3
5  0
6  4
7  4
8  0

```

### `databricks.koalas.groupby.GroupBy.tail`

`GroupBy.tail(n=5) → Union[databricks.koalas.frame.DataFrame, databricks.koalas.series.Series]`

Return last n rows of each group.

Similar to `.apply(lambda x: x.tail(n))`, but it returns a subset of rows from the original DataFrame with original index and order preserved (`as_index` flag is ignored).

Does not work for negative values of n.

#### Returns

**DataFrame or Series**

### Examples

```

>>> df = ks.DataFrame({'a': [1, 1, 1, 1, 2, 2, 2, 3, 3, 3],
...                     'b': [2, 3, 1, 4, 6, 9, 8, 10, 7, 5],
...                     'c': [3, 5, 2, 5, 1, 2, 6, 4, 3, 6]},
...                     columns=['a', 'b', 'c'],
...                     index=[7, 2, 3, 1, 3, 4, 9, 10, 5, 6])
>>> df
   a  b  c
7  1  2  3
2  1  3  5

```

(continues on next page)

(continued from previous page)

```

3   1   1   2
1   1   4   5
3   2   6   1
4   2   9   2
9   2   8   6
10  3  10   4
5   3   7   3
6   3   5   6

```

```

>>> df.groupby('a').tail(2).sort_index()
   a  b  c
1  1  4  5
3  1  1  2
4  2  9  2
5  3  7  3
6  3  5  6
9  2  8  6

```

```

>>> df.groupby('a')['b'].tail(2).sort_index()
1    4
3    1
4    9
5    7
6    5
9    8
Name: b, dtype: int64

```

The following methods are available only for *DataFrameGroupBy* objects.

---

*DataFrameGroupBy.describe()*

Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

---

### **databricks.koalas.groupby.DataFrameGroupBy.describe**

*DataFrameGroupBy.describe()* → *databricks.koalas.frame.DataFrame*

Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

Analyzes both numeric and object series, as well as *DataFrame* column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

---

**Note:** Unlike pandas, the percentiles in Koalas are based upon approximate percentile computation because computing percentiles across a large dataset is extremely expensive.

---

#### **Returns**

**DataFrame** Summary statistics of the *DataFrame* provided.

**See also:**

***DataFrame.count***



`DataFrame.max``DataFrame.min``DataFrame.mean``DataFrame.std`

## Examples

```
>>> df = ks.DataFrame({'a': [1, 1, 3], 'b': [4, 5, 6], 'c': [7, 8, 9]})
>>> df
   a  b  c
0  1  4  7
1  1  5  8
2  3  6  9
```

Describing a DataFrame. By default only numeric fields are returned.

```
>>> described = df.groupby('a').describe()
>>> described.sort_index()
      b                                c
count mean      std min 25% 50% 75% max count mean      std min 25% 50% 75%
a
1    2.0  4.5  0.707107 4.0 4.0 4.0 5.0 5.0    2.0  7.5  0.707107 7.0 7.0 7.0 8.0 8.
3    1.0  6.0      NaN 6.0 6.0 6.0 6.0 6.0    1.0  9.0      NaN 9.0 9.0 9.0 9.0 9.

```

The following methods are available only for *SeriesGroupBy* objects.

<code>SeriesGroupBy.nsmallest([n])</code>	Return the first n rows ordered by columns in ascending order in group.
<code>SeriesGroupBy.nlargest([n])</code>	Return the first n rows ordered by columns in descending order in group.
<code>SeriesGroupBy.value_counts([sort, ...])</code>	Compute group sizes.
<code>SeriesGroupBy.unique()</code>	Return unique values in group.

## `databricks.koalas.groupby.SeriesGroupBy.nsmallest`

`SeriesGroupBy.nsmallest` ( $n=5$ ) → `databricks.koalas.series.Series`

Return the first n rows ordered by columns in ascending order in group.

Return the first n rows with the smallest values in columns, in ascending order. The columns that are not specified are returned as well, but not used for ordering.

### Parameters

**n** [int] Number of items to retrieve.

See also:

`databricks.koalas.Series.nsmallest`

`databricks.koalas.DataFrame.nsmallest`

## Examples

```
>>> df = ks.DataFrame({'a': [1, 1, 1, 2, 2, 2, 3, 3, 3],
...                    'b': [1, 2, 2, 2, 3, 3, 3, 4, 4]}, columns=['a', 'b'])
```

```
>>> df.groupby(['a'])['b'].nsmallest(1).sort_index()
a
1  0    1
2  3    2
3  6    3
Name: b, dtype: int64
```

## databricks.koalas.groupby.SeriesGroupBy.nlargest

`SeriesGroupBy.nlargest` ( $n=5$ ) → `databricks.koalas.series.Series`

Return the first  $n$  rows ordered by columns in descending order in group.

Return the first  $n$  rows with the smallest values in columns, in descending order. The columns that are not specified are returned as well, but not used for ordering.

### Parameters

**n** [int] Number of items to retrieve.

See also:

`databricks.koalas.Series.nlargest`

`databricks.koalas.DataFrame.nlargest`

## Examples

```
>>> df = ks.DataFrame({'a': [1, 1, 1, 2, 2, 2, 3, 3, 3],
...                    'b': [1, 2, 2, 2, 3, 3, 3, 4, 4]}, columns=['a', 'b'])
```

```
>>> df.groupby(['a'])['b'].nlargest(1).sort_index()
a
1  1    2
2  4    3
3  7    4
Name: b, dtype: int64
```

## databricks.koalas.groupby.SeriesGroupBy.value\_counts

`SeriesGroupBy.value_counts` (*sort=None*, *ascending=None*, *dropna=True*) → `databricks.koalas.series.Series`

Compute group sizes.

### Parameters

**sort** [boolean, default None] Sort by frequencies.

**ascending** [boolean, default False] Sort in ascending order.

**dropna** [boolean, default True] Don't include counts of NaN.

See also:

*`databricks.koalas.Series.groupby`*

*`databricks.koalas.DataFrame.groupby`*

## Examples

```
>>> df = ks.DataFrame({'A': [1, 2, 2, 3, 3, 3],
...                    'B': [1, 1, 2, 3, 3, 3]},
...                   columns=['A', 'B'])
>>> df
   A  B
0  1  1
1  2  1
2  2  2
3  3  3
4  3  3
5  3  3
```

```
>>> df.groupby('A')['B'].value_counts().sort_index()
A  B
1  1    1
2  1    1
   2    1
3  3    3
Name: B, dtype: int64
```

## `databricks.koalas.groupby.SeriesGroupBy.unique`

`SeriesGroupBy.unique()` → `databricks.koalas.series.Series`

Return unique values in group.

Uniques are returned in order of unknown. It does NOT sort.

See also:

*`databricks.koalas.Series.unique`*

*`databricks.koalas.Index.unique`*

## Examples

```
>>> df = ks.DataFrame({'a': [1, 1, 1, 2, 2, 2, 3, 3, 3],
...                    'b': [1, 2, 2, 2, 3, 3, 3, 4, 4]},
...                   columns=['a', 'b'])
```

```
>>> df.groupby('a')['b'].unique().sort_index()
a
1    [1, 2]
2    [2, 3]
3    [3, 4]
Name: b, dtype: object
```

## 3.8 Machine Learning utilities

### 3.8.1 MLflow

Arbitrary MLflow models can be used with Koalas Dataframes, provided they implement the ‘pyfunc’ flavor. This is the case for most frameworks supported by MLflow (scikit-learn, pytorch, tensorflow, ...). See comprehensive examples in `load_model()` for more information.

**Note:** The MLflow package must be installed in order to use this module. If MLflow is not installed in your environment already, you can install it with the following command:

```
pip install koalas[mlflow]
```

---

<code>PythonModelWrapper(model_uri, return_type_hint)</code>	re-	A wrapper around MLflow’s Python object model.
<code>load_model(model_uri[, predict_type])</code>		Loads an MLflow model into an wrapper that can be used both for pandas and Koalas DataFrame.

---

#### `databricks.koalas.mlflow.PythonModelWrapper`

**class** `databricks.koalas.mlflow.PythonModelWrapper` (*model\_uri*, *return\_type\_hint*)

A wrapper around MLflow’s Python object model.

This wrapper acts as a predictor on koalas

**\_\_init\_\_** (*model\_uri*, *return\_type\_hint*)

Initialize self. See `help(type(self))` for accurate signature.

#### Methods

---

<code>__init__(model_uri, return_type_hint)</code>	Initialize self.
<code>predict(data)</code>	Returns a prediction on the data.

---

#### `databricks.koalas.mlflow.load_model`

`databricks.koalas.mlflow.load_model` (*model\_uri*, *predict\_type='infer'*) → `databricks.koalas.mlflow.PythonModelWrapper`  
 Loads an MLflow model into an wrapper that can be used both for pandas and Koalas DataFrame.

#### Parameters

**model\_uri** [str] URI pointing to the model. See MLflow documentation for more details.

**predict\_type** [a python basic type, a numpy basic type, a Spark type or ‘infer’.] This is the return type that is expected when calling the predict function of the model. If ‘infer’ is specified, the wrapper will attempt to determine automatically the return type based on the model type.

#### Returns

**PythonModelWrapper** A wrapper around MLflow PythonModel objects. This wrapper is expected to adhere to the interface of `mlflow.pyfunc.PythonModel`.

## Notes

Currently, the model prediction can only be merged back with the existing dataframe. Other columns have to be manually joined. For example, this code will not work:

```
>>> df = ks.DataFrame({"x1": [2.0], "x2": [3.0], "z": [-1]})
>>> features = df[["x1", "x2"]]
>>> y = model.predict(features)
>>> # Works:
>>> features["y"] = y
>>> # Will fail with a message about dataframes not aligned.
>>> df["y"] = y
```

A current workaround is to use the `.merge()` function, using the feature values as merging keys.

```
>>> features['y'] = y
>>> everything = df.merge(features, on=['x1', 'x2'])
>>> everything
   x1  x2  z      y
0  2.0  3.0 -1  1.376932
```

## Examples

Here is a full example that creates a model with scikit-learn and saves the model with MLflow. The model is then loaded as a predictor that can be applied on a Koalas Dataframe.

We first initialize our MLflow environment:

```
>>> from mlflow.tracking import MlflowClient, set_tracking_uri
>>> import mlflow.sklearn
>>> from tempfile import mkdtemp
>>> d = mkdtemp("koalas_mlflow")
>>> set_tracking_uri("file:%s"%d)
>>> client = MlflowClient()
>>> exp = mlflow.create_experiment("my_experiment")
>>> mlflow.set_experiment("my_experiment")
```

We aim at learning this numerical function using a simple linear regressor.

```
>>> from sklearn.linear_model import LinearRegression
>>> train = pd.DataFrame({"x1": np.arange(8), "x2": np.arange(8)**2,
...                      "y": np.log(2 + np.arange(8))})
>>> train_x = train[["x1", "x2"]]
>>> train_y = train[["y"]]
>>> with mlflow.start_run():
...     lr = LinearRegression()
...     lr.fit(train_x, train_y)
...     mlflow.sklearn.log_model(lr, "model")
LinearRegression(...)
```

Now that our model is logged using MLflow, we load it back and apply it on a Koalas dataframe:

```
>>> from databricks.koalas.mlflow import load_model
>>> run_info = client.list_run_infos(exp)[-1]
>>> model = load_model("runs://{run_id}/model".format(run_id=run_info.run_uuid))
>>> prediction_df = ks.DataFrame({"x1": [2.0], "x2": [4.0]})
```

(continues on next page)

(continued from previous page)

```
>>> prediction_df["prediction"] = model.predict(prediction_df)
>>> prediction_df
   x1  x2  prediction
0  2.0  4.0      1.355551
```

The model also works on pandas DataFrames as expected:

```
>>> model.predict(prediction_df[["x1", "x2"]].to_pandas())
array([[1.35555142]])
```

## 3.9 Extensions

### 3.9.1 Accessors

Accessors can be written and registered with Koalas Dataframes, Series, and Index objects. Accessors allow developers to extend the functionality of Koalas objects seamlessly by writing arbitrary classes and methods which are then wrapped in one of the following decorators.

<code>register_dataframe_accessor(name)</code>	Register a custom accessor with a DataFrame
<code>register_series_accessor(name)</code>	Register a custom accessor with a Series object
<code>register_index_accessor(name)</code>	Register a custom accessor with an Index

#### `databricks.koalas.extensions.register_dataframe_accessor`

`databricks.koalas.extensions.register_dataframe_accessor(name)`

Register a custom accessor with a DataFrame

##### Parameters

**name** [str] name used when calling the accessor after its registered

##### Returns

**callable** A class decorator.

See also:

`register_series_accessor` Register a custom accessor on Series objects

`register_index_accessor` Register a custom accessor on Index objects

##### Notes

When accessed, your accessor will be initialized with the Koalas object the user is interacting with. The accessor's init method should always ingest the object being accessed. See the examples for the init signature.

In the pandas API, if data passed to your accessor has an incorrect dtype, it's recommended to raise an `AttributeError` for consistency purposes. In Koalas, `ValueError` is more frequently used to annotate when a value's datatype is unexpected for a given method/function.

Ultimately, you can structure this however you like, but Koalas would likely do something like this:

```
>>> ks.Series(['a', 'b']).dt
...
Traceback (most recent call last):
...
ValueError: Cannot call DatetimeMethods on type StringType
```

## Examples

In your library code:

```
from databricks.koalas.extensions import register_dataframe_accessor

@register_dataframe_accessor("geo")
class GeoAccessor:

    def __init__(self, koalas_obj):
        self._obj = koalas_obj
        # other constructor logic

    @property
    def center(self):
        # return the geographic center point of this DataFrame
        lat = self._obj.latitude
        lon = self._obj.longitude
        return (float(lon.mean()), float(lat.mean()))

    def plot(self):
        # plot this array's data on a map
        pass
```

Then, in an ipython session:

```
>>> ## Import if the accessor is in the other file.
>>> # from my_ext_lib import GeoAccessor
>>> kdf = ks.DataFrame({"longitude": np.linspace(0,10),
...                     "latitude": np.linspace(0, 20)})
>>> kdf.geo.center
(5.0, 10.0)

>>> kdf.geo.plot()
```

## databricks.koalas.extensions.register\_series\_accessor

databricks.koalas.extensions.**register\_series\_accessor**(*name*)

Register a custom accessor with a Series object

### Parameters

**name** [str] name used when calling the accessor after its registered

### Returns

**callable** A class decorator.

See also:

[`register\_dataframe\_accessor`](#) Register a custom accessor on DataFrame objects

**`register_index_accessor`** Register a custom accessor on Index objects

## Notes

When accessed, your accessor will be initialized with the Koalas object the user is interacting with. The code signature must be:

```
def __init__(self, koalas_obj):  
    # constructor logic  
    ...
```

In the pandas API, if data passed to your accessor has an incorrect dtype, it's recommended to raise an `AttributeError` for consistency purposes. In Koalas, `ValueError` is more frequently used to annotate when a value's datatype is unexpected for a given method/function.

Ultimately, you can structure this however you like, but Koalas would likely do something like this:

```
>>> ks.Series(['a', 'b']).dt  
...  
Traceback (most recent call last):  
...  
ValueError: Cannot call DatetimeMethods on type StringType
```

## Examples

In your library code:

```
from databricks.koalas.extensions import register_series_accessor  
  
@register_series_accessor("geo")  
class GeoAccessor:  
  
    def __init__(self, koalas_obj):  
        self._obj = koalas_obj  
  
    @property  
    def is_valid(self):  
        # boolean check to see if series contains valid geometry  
        return True
```

Then, in an ipython session:

```
>>> ## Import if the accessor is in the other file.  
>>> # from my_ext_lib import GeoAccessor  
>>> kdf = ks.DataFrame({"longitude": np.linspace(0,10),  
...                     "latitude": np.linspace(0, 20)})  
>>> kdf.longitude.geo.is_valid  
True
```



**databricks.koalas.extensions.register\_index\_accessor**

`databricks.koalas.extensions.register_index_accessor(name)`

Register a custom accessor with an Index

**Parameters**

**name** [str] name used when calling the accessor after its registered

**Returns**

**callable** A class decorator.

See also:

[`register\_dataframe\_accessor`](#) Register a custom accessor on DataFrame objects

[`register\_series\_accessor`](#) Register a custom accessor on Series objects

**Notes**

When accessed, your accessor will be initialized with the Koalas object the user is interacting with. The code signature must be:

```
def __init__(self, koalas_obj):
    # constructor logic
    ...
```

In the pandas API, if data passed to your accessor has an incorrect dtype, it's recommended to raise an `AttributeError` for consistency purposes. In Koalas, `ValueError` is more frequently used to annotate when a value's datatype is unexpected for a given method/function.

Ultimately, you can structure this however you like, but Koalas would likely do something like this:

```
>>> ks.Series(['a', 'b']).dt
...
Traceback (most recent call last):
...
ValueError: Cannot call DatetimeMethods on type StringType
```

**Examples**

In your library code:

```
from databricks.koalas.extensions import register_index_accessor

@register_index_accessor("foo")
class CustomAccessor:

    def __init__(self, koalas_obj):
        self._obj = koalas_obj
        self.item = "baz"

    @property
    def bar(self):
        # return item value
        return self.item
```

Then, in an ipython session:

```
>>> ## Import if the accessor is in the other file.
>>> # from my_ext_lib import CustomAccessor
>>> kdf = ks.DataFrame({"longitude": np.linspace(0,10),
...                     "latitude": np.linspace(0, 20)})
>>> kdf.index.foo.bar
'baz'
```

## DEVELOPMENT

### 4.1 Contributing Guide

**Table of contents:**

- *Types of Contributions*
- *Step-by-step Guide For Code Contributions*
- *Environment Setup*
- *Running Tests*
- *Building Documentation*
- *Coding Conventions*
- *Doctest Conventions*
- *Release Guide*

#### 4.1.1 Types of Contributions

The largest amount of work consists simply of implementing the pandas API using Spark's built-in functions, which is usually straightforward. But there are many different forms of contributions in addition to writing code:

1. Use the project and provide feedback, by creating new tickets or commenting on existing relevant tickets.
2. Review existing pull requests.
3. Improve the project's documentation.
4. Write blog posts or tutorial articles evangelizing Koalas and help new users learn Koalas.
5. Give a talk about Koalas at your local meetup or a conference.

### 4.1.2 Step-by-step Guide For Code Contributions

1. Read and understand the *Design Principles* for the project. Contributions should follow these principles.
2. Signaling your work: If you are working on something, comment on the relevant ticket that you are doing so to avoid multiple people taking on the same work at the same time. It is also a good practice to signal that your work has stalled or you have moved on and want somebody else to take over.
3. Understand what the functionality is in pandas or in Spark.
4. Implement the functionality, with test cases providing close to 100% statement coverage. Document the functionality.
5. Run existing and new test cases to make sure they still pass. Also run *dev/reformat* script to reformat Python files by using *Black*, and run the linter *dev/lint-python*.
6. Build the docs (*make html* in *docs* directory) and verify the docs related to your change look OK.
7. Submit a pull request, and be responsive to code review feedback from other community members.

That's it. Your contribution, once merged, will be available in the next release.

### 4.1.3 Environment Setup

#### Conda

If you are using Conda, the Koalas installation and development environment are as follows.

```
# Python 3.6+ is required
conda create --name koalas-dev-env python=3.6
conda activate koalas-dev-env
conda install -c conda-forge pyspark=2.4
pip install -r requirements-dev.txt
pip install -e . # installs koalas from current checkout
```

Once setup, make sure you switch to *koalas-dev-env* before development:

```
conda activate koalas-dev-env
```

#### pip

With Python 3.6+, pip can be used as below to install and set up the development environment.

```
pip install pyspark==2.4
pip install -r requirements-dev.txt
pip install -e . # installs koalas from current checkout
```

### 4.1.4 Running Tests

There is a script `./dev/pytest` which is exactly same as `pytest` but with some default settings to run Koalas tests easily.

To run all the tests, similar to our CI pipeline:

```
# Run all unittest and doctest
./dev/pytest
```

To run a specific test file:

```
# Run unittest
./dev/pytest -k test_dataframe.py

# Run doctest
./dev/pytest -k series.py --doctest-modules databricks
```

To run a specific doctest/unittest:

```
# Run unittest
./dev/pytest -k "DataFrameTest and test_Dataframe"

# Run doctest
./dev/pytest -k DataFrame.corr --doctest-modules databricks
```

Note that `-k` is used for simplicity although it takes an expression. You can use `-verbose` to check what to filter. See `pytest -help` for more details.

### 4.1.5 Building Documentation

To build documentation via Sphinx:

```
cd docs && make clean html
```

It generates HTMLs under `docs/build/html` directory. Open `docs/build/html/index.html` to check if documentation is built properly.

### 4.1.6 Coding Conventions

We follow [PEP 8](#) with one exception: lines can be up to 100 characters in length, not 79.

### 4.1.7 Doctest Conventions

When writing doctests, usually the doctests in pandas are converted into Koalas to make sure the same codes work in Koalas. In general, doctests should be grouped logically by separating a newline.

For instance, the first block is for the statements for preparation, the second block is for using the function with a specific argument, and third block is for another argument. As a example, please refer `DataFrame.rsub` in pandas.

These blocks should be consistently separated in Koalas, and more doctests should be added if the coverage of the doctests or the number of examples to show is not enough even though they are different from pandas’.

## 4.1.8 Release Guide

### Release Cadence

Koalas 1.8.0 is the last minor release because Koalas will be officially included to PySpark. There will be only maintenance releases. Users are expected to directly use PySpark with Apache Spark 3.2+.

### Release Instructions

Only project maintainers can do the following to publish a release.

1. Make sure version is set correctly in *databricks/koalas/version.py*.
2. Make sure the build is green.
3. Create a new release on GitHub. Tag it as the same version as the setup.py. If the version is “0.1.0”, tag the commit as “v0.1.0”.
4. Upload the package to PyPi:

```
rm -rf dist/koalas*
python setup.py sdist bdist_wheel
export package_version=$(python setup.py --version)
echo $package_version

python3 -m pip install --user --upgrade twine

# for test
python3 -m twine upload --repository-url https://test.pypi.org/legacy/ dist/
↪koalas-$package_version-py3-none-any.whl dist/koalas-$package_version.tar.
↪gz

# for release
python3 -m twine upload --repository-url https://upload.pypi.org/legacy/_
↪dist/koalas-$package_version-py3-none-any.whl dist/koalas-$package_version.
↪tar.gz
```

5. Verify the uploaded package can be installed and executed. One unofficial tip is to run the doctests of Koalas within a Python interpreter after installing it.

```
import os

from pytest import main
import databricks

test_path = os.path.abspath(os.path.dirname(databricks.__file__))
main(['-k', '-to_delta -read_delta', '--verbose', '--showlocals', '--doctest-
↪modules', test_path])
```

Note that this way might require additional settings, for instance, environment variables.

## 4.2 Design Principles

This section outlines design principles guiding the Koalas project.

### 4.2.1 Be Pythonic

Koalas targets Python data scientists. We want to stick to the convention that users are already familiar with as much as possible. Here are some examples:

- Function names and parameters use snake\_case, rather than CamelCase. This is different from PySpark's design. For example, Koalas has `to_pandas()`, whereas PySpark has `toPandas()` for converting a DataFrame into a pandas DataFrame. In limited cases, to maintain compatibility with Spark, we also provide Spark's variant as an alias.
- Koalas respects to the largest extent the conventions of the Python numerical ecosystem, and allows the use of NumPy types, etc. that can be supported by Spark.
- Koalas docs' style and infrastructure simply follow rest of the PyData projects'.

### 4.2.2 Unify small data (pandas) API and big data (Spark) API, but pandas first

The Koalas DataFrame is meant to provide the best of pandas and Spark under a single API, with easy and clear conversions between each API when necessary. When Spark and pandas have similar APIs with subtle differences, the principle is to honor the contract of the pandas API first.

There are different classes of functions:

1. Functions that are found in both Spark and pandas under the same name (*count*, *dtypes*, *head*). The return value is the same as the return type in pandas (and not Spark's).
2. Functions that are found in Spark but that have a clear equivalent in pandas, e.g. *alias* and *rename*. These functions will be implemented as the alias of the pandas function, but should be marked that they are aliases of the same functions. They are provided so that existing users of PySpark can get the benefits of Koalas without having to adapt their code.
3. Functions that are only found in pandas. When these functions are appropriate for distributed datasets, they should become available in Koalas.
4. Functions that are only found in Spark that are essential to controlling the distributed nature of the computations, e.g. *cache*. These functions should be available in Koalas.

We are still debating whether data transformation functions only available in Spark should be added to Koalas, e.g. *select*. We would love to hear your feedback on that.

### 4.2.3 Return Koalas data structure for big data, and pandas data structure for small data

Often developers face the question whether a particular function should return a Koalas DataFrame/Series, or a pandas DataFrame/Series. The principle is: if the returned object can be large, use a Koalas DataFrame/Series. If the data is bound to be small, use a pandas DataFrame/Series. For example, `DataFrame.dtypes` return a pandas Series, because the number of columns in a DataFrame is bounded and small, whereas `DataFrame.head()` or `Series.unique()` returns a Koalas DataFrame/Series, because the resulting object can be large.

#### 4.2.4 Provide discoverable APIs for common data science tasks

At the risk of overgeneralization, there are two API design approaches: the first focuses on providing APIs for common tasks; the second starts with abstractions, and enable users to accomplish their tasks by composing primitives. While the world is not black and white, pandas takes more of the former approach, while Spark has taken more of the later.

One example is value count (count by some key column), one of the most common operations in data science. pandas `DataFrame.value_count` returns the result in sorted order, which in 90% of the cases is what users prefer when exploring data, whereas Spark's does not sort, which is more desirable when building data pipelines, as users can accomplish the pandas behavior by adding an explicit `orderBy`.

Similar to pandas, Koalas should also lean more towards the former, providing discoverable APIs for common data science tasks. In most cases, this principle is well taken care of by simply implementing pandas' APIs. However, there will be circumstances in which pandas' APIs don't address a specific need, e.g. plotting for big data.

#### 4.2.5 Provide well documented APIs, with examples

All functions and parameters should be documented. Most functions should be documented with examples, because those are the easiest to understand than a blob of text explaining what the function does.

A recommended way to add documentation is to start with the docstring of the corresponding function in PySpark or pandas, and adapt it for Koalas. If you are adding a new function, also add it to the API reference doc index page in `docs/source/reference` directory. The examples in docstring also improve our test coverage.

#### 4.2.6 Guardrails to prevent users from shooting themselves in the foot

Certain operations in pandas are prohibitively expensive as data scales, and we don't want to give users the illusion that they can rely on such operations in Koalas. That is to say, methods implemented in Koalas should be safe to perform by default on large datasets. As a result, the following capabilities are not implemented in Koalas:

1. Capabilities that are fundamentally not parallelizable: e.g. imperatively looping over each element
2. Capabilities that require materializing the entire working set in a single node's memory. This is why we do not implement `pandas.DataFrame.to_xarray`. Another example is the `_repr_html_` call caps the total number of records shown to a maximum of 1000, to prevent users from blowing up their driver node simply by typing the name of the DataFrame in a notebook.

A few exceptions, however, exist. One common pattern with "big data science" is that while the initial dataset is large, the working set becomes smaller as the analysis goes deeper. For example, data scientists often perform aggregation on datasets and want to then convert the aggregated dataset to some local data structure. To help data scientists, we offer the following:

- `DataFrame.to_pandas()`: returns a pandas DataFrame, koalas only
- `DataFrame.to_numpy()`: returns a numpy array, works with both pandas and Koalas

Note that it is clear from the names that these functions return some local data structure that would require materializing data in a single node's memory. For these functions, we also explicitly document them with a warning note that the resulting data structure must be small.



### 4.2.7 Be a lean API layer and move fast

Koalas is designed as an API overlay layer on top of Spark. The project should be lightweight, and most functions should be implemented as wrappers around Spark or pandas - the Koalas library is designed to be used only in the Spark's driver side in general. Koalas does not accept heavyweight implementations, e.g. execution engine changes.

This approach enables us to move fast. For the considerable future, we aim to be making monthly releases. If we find a critical bug, we will be making a new release as soon as the bug fix is available.

### 4.2.8 High test coverage

Koalas should be well tested. The project tracks its test coverage with over 90% across the entire codebase, and close to 100% for critical parts. Pull requests will not be accepted unless they have close to 100% statement coverage from the codecov report.



## RELEASE NOTES

### 5.1 Version 1.8.2

Koalas 1.8.2 is a maintenance release. Koalas is [officially included in PySpark as `pandas API on Spark` in Apache Spark 3.2](#). In Apache Spark 3.2+, please use Apache Spark directly.

Although moving to pandas API on Spark is recommended, Koalas 1.8.2 still works with Spark 3.2 (#2203).

#### 5.1.1 Improvements and bug fixes

- `_builtin_table` import in `groupby apply` (changed in `pandas`  $\geq 1.3.0$ ). (#2184)

### 5.2 Version 1.8.1

Koalas 1.8.1 is a maintenance release. Koalas will be [officially included in PySpark in the upcoming Apache Spark 3.2](#). In Apache Spark 3.2+, please use Apache Spark directly.

#### 5.2.1 Improvements and bug fixes

- Remove the upperbound for `numpy`. (#2166)
- Allow Python 3.9 when the underlying PySpark is 3.1 and above. (#2167)

Along with the following fixes: - Support `x` and `y` properly in plots (both `matplotlib` and `plotly`). (#2172) - Fix `Index.different` to work properly. (#2173) - Fix backward compatibility for Python version 3.5.\*. (#2174)

### 5.3 Version 1.8.0

Koalas 1.8.0 is the last minor release because Koalas will be [officially included in PySpark in the upcoming Apache Spark 3.2](#). In Apache Spark 3.2+, please use Apache Spark directly.

### 5.3.1 Categorical type and ExtensionDtype

We added the support of pandas' categorical type (#2064, #2106).

```
>>> s = ks.Series(list("abbccc"), dtype="category")
>>> s
0    a
1    b
2    b
3    c
4    c
5    c
dtype: category
Categories (3, object): ['a', 'b', 'c']
>>> s.cat.categories
Index(['a', 'b', 'c'], dtype='object')
>>> s.cat.codes
0    0
1    1
2    1
3    2
4    2
5    2
dtype: int8
>>> idx = ks.CategoricalIndex(list("abbccc"))
>>> idx
CategoricalIndex(['a', 'b', 'b', 'c', 'c', 'c'],
                  categories=['a', 'b', 'c'], ordered=False, dtype='category')

>>> idx.codes
Int64Index([0, 1, 1, 2, 2, 2], dtype='int64')
>>> idx.categories
Index(['a', 'b', 'c'], dtype='object')
```

and `ExtensionDtype` as type arguments to annotate return types (#2120, #2123, #2132, #2127, #2126, #2125, #2124):

```
def func() -> ks.Series[pd.Int32Dtype()]:
    ...
```

### 5.3.2 Other new features, improvements and bug fixes

We added the following new features:

**DataFrame:**

- `first` (#2128)
- `at_time` (#2116)

**Series:**

- `at_time` (#2130)
- `first` (#2128)
- `between_time` (#2129)

**DatetimeIndex:**

- `indexer_between_time` (#2104)

- `indexer_at_time` (#2109)
- `between_time` (#2111)

Along with the following fixes:

- Support tuple to `(DataFrame|Series).replace()` (#2095)
- Check `index_dtype` and `data_dtypes` more strictly. (#2100)
- Return actual values via `toPandas`. (#2077)
- Add `lines` and `orient` to `read_json` and `to_json` to improve error message (#2110)
- Fix `isin` to accept numpy array (#2103)
- Allow multi-index column names for inferring return type schema with names. (#2117)
- Add a short JDBC user guide (#2148)
- Remove upper bound pandas 1.2 (#2141)
- Standardize exceptions of arithmetic operations on Datetime-like data (#2101)

## 5.4 Version 1.7.0

### 5.4.1 Switch the default plotting backend to Plotly

We switched the default plotting backend from Matplotlib to Plotly (#2029, #2033). In addition, we added more Plotly methods such as `DataFrame.plot.kde` and `Series.plot.kde` (#2028).

```
import databricks.koalas as ks
kdf = ks.DataFrame({
    'a': [1, 2, 2.5, 3, 3.5, 4, 5],
    'b': [1, 2, 3, 4, 5, 6, 7],
    'c': [0.5, 1, 1.5, 2, 2.5, 3, 3.5]})
kdf.plot.hist()
```

Plotting backend can be switched to matplotlib by setting `ks.options.plotting.backend` to `matplotlib`.

```
ks.options.plotting.backend = "matplotlib"
```

### 5.4.2 Add Int64Index, Float64Index, DatetimeIndex

We added more types of Index such as `Int64Index`, `Float64Index` and `DatetimeIndex` (#2025, #2066).

When creating an index, Index instance is always returned regardless of the data type.

But now `Int64Index`, `Float64Index` or `DatetimeIndex` is returned depending on the data type of the index.

```
>>> type(ks.Index([1, 2, 3]))
<class 'databricks.koalas.indexes.numeric.Int64Index'>
>>> type(ks.Index([1.1, 2.5, 3.0]))
<class 'databricks.koalas.indexes.numeric.Float64Index'>
>>> type(ks.Index([datetime.datetime(2021, 3, 9)]))
<class 'databricks.koalas.indexes.datetimes.DatetimeIndex'>
```

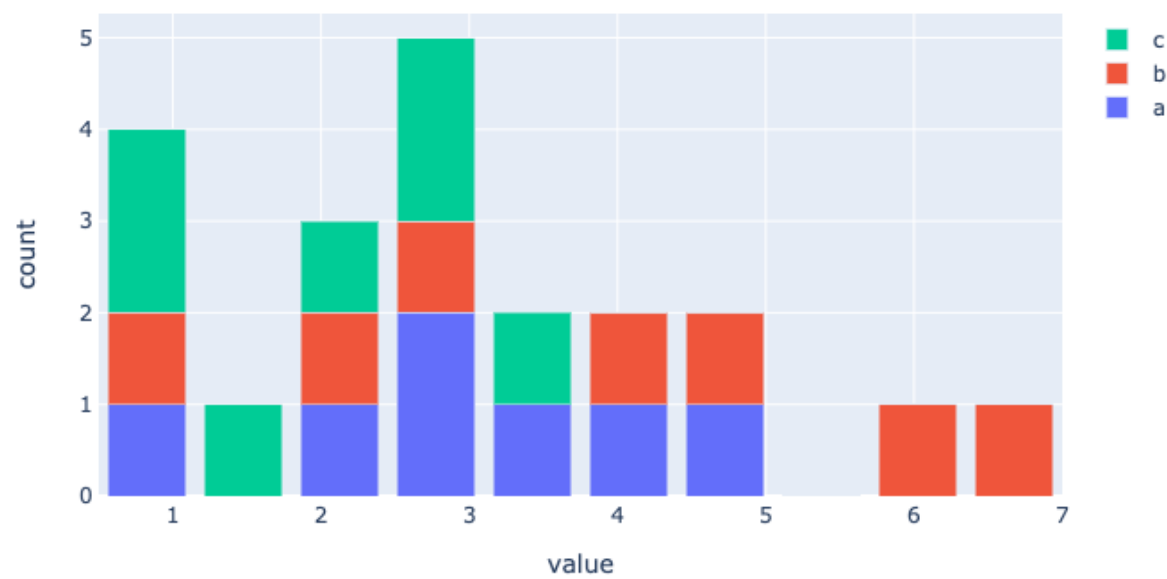


Fig. 1: Koalas\_plotly\_hist\_plot

In addition, we added many properties for `DatetimeIndex` such as `year`, `month`, `day`, `hour`, `minute`, `second`, etc. (#2074) and added APIs for `DatetimeIndex` such as `round()`, `floor()`, `ceil()`, `normalize()`, `strftime()`, `month_name()` and `day_name()` (#2082, #2086, #2089).

### 5.4.3 Create Index from Series or Index objects

Index can be created by taking Series or Index objects (#2071).

```
>>> kser = ks.Series([1, 2, 3], name="a", index=[10, 20, 30])
>>> ks.Index(kser)
Int64Index([1, 2, 3], dtype='int64', name='a')
>>> ks.Int64Index(kser)
Int64Index([1, 2, 3], dtype='int64', name='a')
>>> ks.Float64Index(kser)
Float64Index([1.0, 2.0, 3.0], dtype='float64', name='a')
```

```
>>> kser = ks.Series([datetime(2021, 3, 1), datetime(2021, 3, 2)], index=[10, 20])
>>> ks.Index(kser)
DatetimeIndex(['2021-03-01', '2021-03-02'], dtype='datetime64[ns]', freq=None)
>>> ks.DatetimeIndex(kser)
DatetimeIndex(['2021-03-01', '2021-03-02'], dtype='datetime64[ns]', freq=None)
```

### 5.4.4 Extension dtypes support

We added basic extension dtypes support (#2039).

```
>>> kdf = ks.DataFrame(
...     {
...         "a": [1, 2, None, 3],
...         "b": [4.5, 5.2, 6.1, None],
...         "c": ["A", "B", "C", None],
...         "d": [False, None, True, False],
...     }
... ).astype({"a": "Int32", "b": "Float64", "c": "string", "d": "boolean"})
>>> kdf
   a    b    c    d
0  1  4.5    A  False
1  2  5.2    B  <NA>
2  <NA> 6.1    C   True
3  3  NaN  <NA>  False
>>> kdf.dtypes
a      Int32
b    float64
c      string
d     boolean
dtype: object
```

The following types are supported per the installed pandas:

- pandas >= 0.24
- `Int8Dtype`
- `Int16Dtype`
- `Int32Dtype`

- Int64Dtype
- pandas >= 1.0
- BooleanDtype
- StringDtype
- pandas >= 1.2
- Float32Dtype
- Float64Dtype

Binary operations and type casting are supported:

```
>>> kdf.a + kdf.b
0      5
1      7
2    <NA>
3    <NA>
dtype: Int64
>>> kdf + kdf
      a      b
0     2     8
1     4    10
2  <NA>    12
3     6  <NA>
>>> kdf.a.astype('Float64')
0     1.0
1     2.0
2    <NA>
3     3.0
Name: a, dtype: Float64
```

### 5.4.5 Other new features, improvements and bug fixes

We added the following new features:

koalas:

- `date_range` (#2081)
- `read_orc` (#2017)

Series:

- `align` (#2019)

DataFrame:

- `align` (#2019)
- `to_orc` (#2024)

Along with the following fixes:

- PySpark 3.1.1 Support
- Preserve index for statistical functions with `axis==1` (#2036)
- Use `iloc` to make sure it retrieves the first element (#2037)
- Fix `numeric_only` to follow pandas (#2035)

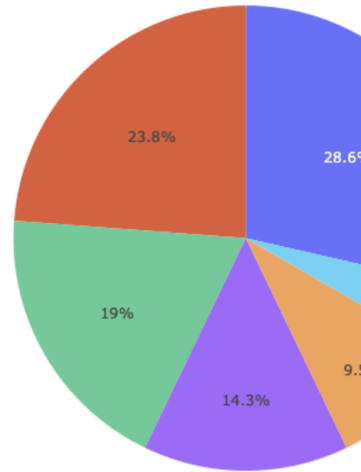


- Fix DataFrame.merge to work properly (#2060)
- Fix astype(str) for some data types (#2040)
- Fix binary operations Index by Series (#2046)
- Fix bug on pow and rpow (#2047)
- Support bool list-like column selection for loc indexer (#2057)
- Fix window functions to resolve (#2090)
- Refresh GitHub workflow matrix (#2083)
- Restructure the hierarchy of Index unit tests (#2080)
- Fix to delegate dtypes (#2061)

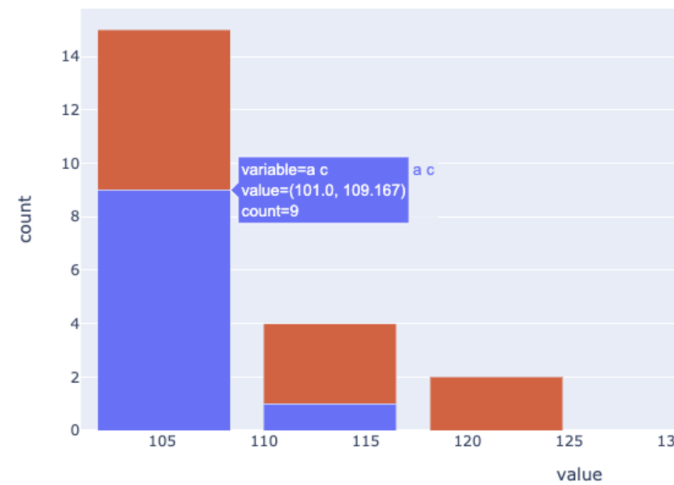
## 5.5 Version 1.6.0

### 5.5.1 Improved Plotly backend support

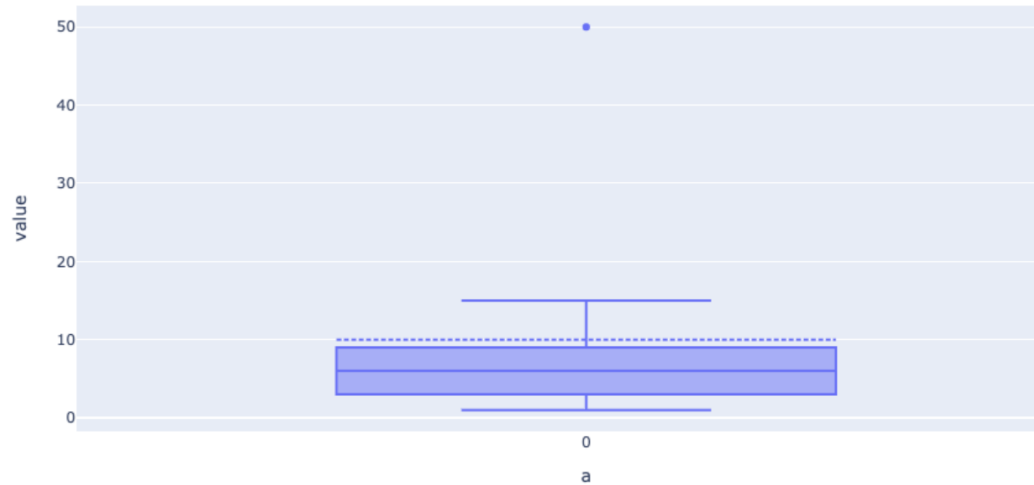
We improved plotting support by implementing pie, histogram and box plots with Plotly plot backend. Koalas now can plot data with Plotly via:



- DataFrame.plot.pie and Series.plot.pie (#1971)



- `DataFrame.plot.hist` and `Series.plot.hist` (#1999)



- `Series.plot.box` (#2007)

In addition, we optimized histogram calculation as a single pass in `DataFrame` (#1997) instead of launching each job to calculate each `Series` in `DataFrame`.

## 5.5.2 Operations between Series and Index

The operations between `Series` and `Index` are now supported as below (#1996):

```
>>> kser = ks.Series([1, 2, 3, 4, 5, 6, 7])
>>> kidx = ks.Index([0, 1, 2, 3, 4, 5, 6])

>>> (kser + 1 + 10 * kidx).sort_index()
0      2
1     13
2     24
3     35
4     46
5     57
6     68
```

(continues on next page)

(continued from previous page)

```

dtype: int64
>>> (kidx + 1 + 10 * kser).sort_index()
0      11
1      22
2      33
3      44
4      55
5      66
6      77
dtype: int64

```

### 5.5.3 Support setting to a Series via attribute access

We have added the support of setting a column via attribute assignment in DataFrame, (#1989).

```

>>> kdf = ks.DataFrame({'A': [1, 2, 3, None]})
>>> kdf.A = kdf.A.fillna(kdf.A.median())
>>> kdf
   A
0  1.0
1  2.0
2  3.0
3  2.0

```

### 5.5.4 Other new features, improvements and bug fixes

We added the following new features:

Series:

- factorize (#1972)
- sem (#1993)

DataFrame

- insert (#1983)
- sem (#1993)

In addition, we also implement new parameters:

- Add min\_count parameter for Frame.sum. (#1978)
- Added ddof parameter for GroupBy.std() and GroupBy.var() (#1994)
- Support ddof parameter for std and var. (#1986)

Along with the following fixes:

- Fix stat functions with no numeric columns. (#1967)
- Fix DataFrame.replace with NaN/None values (#1962)
- Fix cumsum and cumprod. (#1982)
- Use Python type name instead of Spark's in error messages. (#1985)
- Use object.\_\_setattr\_\_ in Series. (#1991)

- Adjust Series.mode to match pandas Series.mode (#1995)
- Adjust data when all the values in a column are nulls. (#2004)
- Fix as\_spark\_type to not support “bigint”. (#2011)

## 5.6 Version 1.5.0

### 5.6.1 Index operations support

We improved Index operations support (#1944, #1955).

Here are some examples:

- Before

```
>>> kidx = ks.Index([1, 2, 3, 4, 5])
>>> kidx + kidx
Int64Index([2, 4, 6, 8, 10], dtype='int64')
>>> kidx + kidx + kidx
Traceback (most recent call last):
...
AssertionError: args should be single DataFrame or single/multiple Series
```

```
>>> ks.Index([1, 2, 3, 4, 5]) + ks.Index([6, 7, 8, 9, 10])
Traceback (most recent call last):
...
AssertionError: args should be single DataFrame or single/multiple Series
```

- After

```
>>> kidx = ks.Index([1, 2, 3, 4, 5])
>>> kidx + kidx + kidx
Int64Index([3, 6, 9, 12, 15], dtype='int64')
```

```
>>> ks.options.compute.ops_on_diff_frames = True
>>> ks.Index([1, 2, 3, 4, 5]) + ks.Index([6, 7, 8, 9, 10])
Int64Index([7, 9, 13, 11, 15], dtype='int64')
```

### 5.6.2 Other new features and improvements

We added the following new features:

DataFrame:

- swaplevel (#1928)
- swapaxes (#1946)
- dot (#1945)
- itertuples (#1960)

Series:

- swaplevel (#1919)
- swapaxes (#1954)

Index:

- `to_list` (#1948)

MultiIndex:

- `to_list` (#1948)

GroupBy: `-tail` (#1949) - `median` (#1957)

### 5.6.3 Other improvements and bug fixes

- Support DataFrame parameter in `Series.dot` (#1931)
- Add a best practice for checkpointing. (#1930)
- Remove implicit switch-ons of “`compute.ops_on_diff_frames`” (#1953)
- Fix `Series._to_internal_pandas` and introduce `Index._to_internal_pandas`. (#1952)
- Fix `first/last_valid_index` to support empty column DataFrame. (#1923)
- Use pandas’ transpose when the data is expected to be small. (#1932)
- Fix `tail` to use the resolved copy (#1942)
- Avoid unneeded `reset_index` in `DataFrameGroupBy.describe`. (#1951)
- `TypeError` when `Index.name` / `Series.name` is not a hashable type (#1883)
- Adjust data column names before attaching default index. (#1947)
- Add `plotly` into the optional dependency in Koalas (#1939)
- Add `plotly` backend test cases (#1938)
- Don’t pass stacked in `plotly` area chart (#1934)
- Set upperbound of `matplotlib` to avoid failure on Ubuntu (#1959)
- Fix `GroupBy.describe` for multi-index columns. (#1922)
- Upgrade pandas version in CI (#1961)
- Compare Series from the same anchor (#1956)
- Add videos from Data+AI Summit 2020 EUROPE. (#1963)
- Set `PYARROW_IGNORE_TIMEZONE` for binder. (#1965)

## 5.7 Version 1.4.0

### 5.7.1 Better type support

We improved the type mapping between pandas and Koalas (#1870, #1903). We added more types or string expressions to specify the data type or fixed mismatches between pandas and Koalas.

Here are some examples:

- Added `np.float32` and `"float32"` (matched to `FloatType`)

```
>>> ks.Series([10]).astype(np.float32)
0    10.0
dtype: float32

>>> ks.Series([10]).astype("float32")
0    10.0
dtype: float32
```

- Added `np.datetime64` and `"datetime64[ns]"` (matched to `TimestampType`)

```
>>> ks.Series(["2020-10-26"]).astype(np.datetime64)
0    2020-10-26
dtype: datetime64[ns]

>>> ks.Series(["2020-10-26"]).astype("datetime64[ns]")
0    2020-10-26
dtype: datetime64[ns]
```

- Fixed `np.int` to match `LongType`, not `IntegerType`.

```
>>> pd.Series([100]).astype(np.int)
0    100.0
dtype: int64

>>> ks.Series([100]).astype(np.int)
0    100.0
dtype: int32 # This fixed to `int64` now.
```

- Fixed `np.float` to match `DoubleType`, not `FloatType`.

```
>>> pd.Series([100]).astype(np.float)
0    100.0
dtype: float64

>>> ks.Series([100]).astype(np.float)
0    100.0
dtype: float32 # This fixed to `float64` now.
```

We also added a document which describes supported/unsupported pandas data types or data type mapping between pandas data types and PySpark data types. See: [Type Support In Koalas](#).

## 5.7.2 Return type annotations for major Koalas objects

To improve Koala's auto-completion in various editors and avoid misuse of APIs, we added return type annotations to major Koalas objects. These objects include `DataFrame`, `Series`, `Index`, `GroupBy`, `Window` objects, etc. (#1852, #1857, #1859, #1863, #1871, #1882, #1884, #1889, #1892, #1894, #1898, #1899, #1900, #1902).

The return type annotations help auto-completion libraries, such as `Jedi`, to infer the actual data type and provide proper suggestions:

- Before
- After

It also helps mypy enable static analysis over the method body.

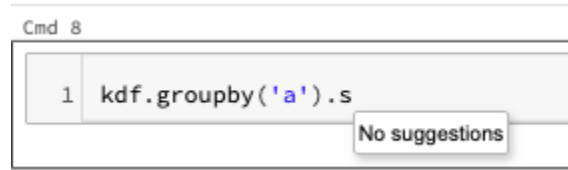


Fig. 2: Before

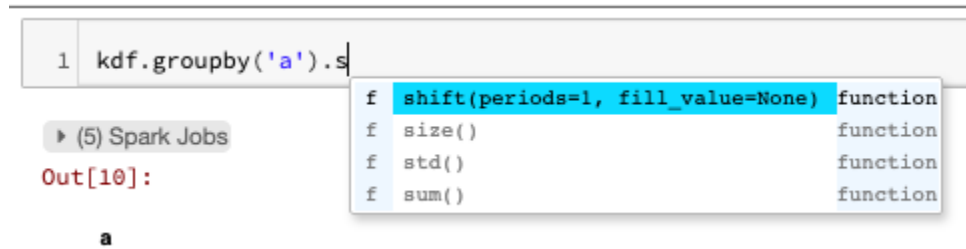


Fig. 3: After

### 5.7.3 pandas 1.1.4 support

We verified the behaviors of pandas 1.1.4 in Koalas.

As pandas 1.1.4 introduced a behavior change related to `MultiIndex.is_monotonic` (`MultiIndex.is_monotonic_increasing`) and `MultiIndex.is_monotonic_decreasing` (pandas-dev/pandas#37220), Koalas also changes the behavior (#1881).

### 5.7.4 Other new features and improvements

We added the following new features:

DataFrame:

- `__neg__` (#1847)
- `rename_axis` (#1843)
- `spark.repartition` (#1864)
- `spark.coalesce` (#1873)
- `spark.checkpoint` (#1877)
- `spark.local_checkpoint` (#1878)
- `reindex_like` (#1880)

Series:

- `rename_axis` (#1843)
- `compare` (#1802)
- `reindex_like` (#1880)

Index:

- `intersection` (#1747)

MultiIndex:

- `intersection` (#1747)

## 5.7.5 Other improvements and bug fixes

- Use `SF.repeat` in `series.str.repeat` (#1844)
- Remove warning when use cache in the context manager (#1848)
- Support a non-string name in Series' boxplot (#1849)
- Calculate fliers correctly in `Series.plot.box` (#1846)
- Show type name rather than type class in error messages (#1851)
- Fix `DataFrame.spark.hint` to reflect internal changes. (#1865)
- `DataFrame.reindex` supports named columns index (#1876)
- Separate `InternalFrame.index_map` into `index_spark_column_names` and `index_names`. (#1879)
- Fix `DataFrame.xs` to handle internal changes properly. (#1896)
- Explicitly disallow empty list as `index_spark_colum_names` and `index_names`. (#1895)
- Use nullable inferred schema in function apply (#1897)
- Introduce `InternalFrame.index_level`. (#1890)
- Remove `InternalFrame.index_map`. (#1901)
- Force to use the Spark's system default precision and scale when inferred data type contains `DecimalType`. (#1904)
- Upgrade PyArrow from 1.0.1 to 2.0.0 in CI (#1860)
- Fix `read_excel` to support squeeze argument. (#1905)
- Fix `to_csv` to avoid duplicated option 'path' for `DataFrameWriter`. (#1912)

## 5.8 Version 1.3.0

### 5.8.1 pandas 1.1 support

We verified the behaviors of pandas 1.1 in Koalas. Koalas now supports pandas 1.1 officially (#1688, #1822, #1829).

### 5.8.2 Support for non-string names

Now we support for non-string names (#1784). Previously names in Koalas, e.g., `df.columns`, `df.columns.names`, `df.index.names`, needed to be a string or a tuple of string, but it should allow other data types which are supported by Spark.

**Before:**

```
>>> kdf = ks.DataFrame([[1, 'x'], [2, 'y'], [3, 'z']])
>>> kdf.columns
Index(['0', '1'], dtype='object')
```

**After:**



```
>>> kdf = ks.DataFrame([[1, 'x'], [2, 'y'], [3, 'z']])
>>> kdf.columns
Int64Index([0, 1], dtype='int64')
```

### 5.8.3 Improve distributed-sequence default index

The performance is improved when creating a distributed-sequence as a default index type by avoiding the interaction between Python and JVM (#1699).

### 5.8.4 Standardize binary operations between int and str columns

Make behaviors of binary operations (+, -, \*, /, //, %) between `int` and `str` columns consistent with respective pandas behaviors (#1828).

It standardizes binary operations as follows:

- `+`: raise `TypeError` between `int` column and `str` column (or string literal)
- `*`: act as spark SQL `repeat` between `int` column(or int literal) and `str` columns; raise `TypeError` if a string literal is involved
- `-`, `/`, `//`, `%` (modulo): raise `TypeError` if a `str` column (or string literal) is involved

### 5.8.5 Other new features and improvements

We added the following new features:

`DataFrame`:

- `product` (#1739)
- `from_dict` (#1778)
- `pad` (#1786)
- `backfill` (#1798)

`Series`:

- `reindex` (#1737)
- `explode` (#1777)
- `pad` (#1786)
- `argmin` (#1790)
- `argmax` (#1790)
- `argsort` (#1793)
- `backfill` (#1798)

`Index`:

- `inferred_type` (#1745)
- `item` (#1744)
- `is_unique` (#1766)

- `asi8` (#1764)
- `is_type_compatible` (#1765)
- `view` (#1788)
- `insert` (#1804)

**MultiIndex:**

- `inferred_type` (#1745)
- `item` (#1744)
- `is_unique` (#1766)
- `asi8` (#1764)
- `is_type_compatible` (#1765)
- `from_frame` (#1762)
- `view` (#1788)
- `insert` (#1804)

**GroupBy:**

- `get_group` (#1783)

### 5.8.6 Other improvements

- Fix `DataFrame.mad` to work properly (#1749)
- Fix Series name after binary operations. (#1753)
- Fix `GroupBy.cum~` for matching with pandas' behavior (#1708)
- Fix `cumprod` to work properly with Integer columns. (#1750)
- Fix `DataFrame.join` for MultiIndex (#1771)
- Exception handling for `from_frame` properly (#1791)
- Fix `iloc` for `slice(None, 0)` (#1767)
- Fix Series `__repr__` when Series.name is None. (#1796)
- `DataFrame.reindex` supports koalas Index parameter (#1741)
- Fix Series `fillna` with `inplace=True` on non-nullable column. (#1809)
- Input check in various APIs (#1808, #1810, #1811, #1812, #1813, #1814, #1816, #1824)
- Fix `to_list` work properly in pandas==0.23 (#1823)
- Fix Series `astype` to work properly (#1818)
- `Frame.groupby` supports `dropna` (#1815)

## 5.9 Version 1.2.0

### 5.9.1 Non-named Series support

Now we added support for non-named Series ([#1712](#)). Previously Koalas automatically named a Series "0" if no name is specified or None is set to the name, whereas pandas allows a Series without the name.

For example:

```
>>> ks.__version__
'1.1.0'
>>> kser = ks.Series([1, 2, 3])
>>> kser
0    1
1    2
2    3
Name: 0, dtype: int64
>>> kser.name = None
>>> kser
0    1
1    2
2    3
Name: 0, dtype: int64
```

Now the Series will be non-named.

```
>>> ks.__version__
'1.2.0'
>>> ks.Series([1, 2, 3])
0    1
1    2
2    3
dtype: int64
>>> kser = ks.Series([1, 2, 3], name="a")
>>> kser.name = None
>>> kser
0    1
1    2
2    3
dtype: int64
```

### 5.9.2 More stable “distributed-sequence” default index

Previously “distributed-sequence” default index had sometimes produced wrong values or even raised an exception. For example, the codes below:

```
>>> from databricks import koalas as ks
>>> ks.options.compute.default_index_type = 'distributed-sequence'
>>> ks.range(10).reset_index()
```

did not work as below:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
```

(continues on next page)

(continued from previous page)

```
pyspark.sql.utils.PythonException:
  An exception was thrown from the Python worker. Please see the stack trace below.
Traceback (most recent call last):
  ...
  File ".../koalas/databricks/koalas/internal.py", line 620, in offset
    current_partition_offset = sums[id.iloc[0]]
KeyError: 103
```

We investigated and made the default index type more stable (#1701). Now it unlikely causes such situations and it is stable enough.

### 5.9.3 Improve testing infrastructure

We changed the testing infrastructure to use pandas' testing utils for exact check (#1722). Now it compares even index/column types and names so that we will be able to follow pandas more strictly.

### 5.9.4 Other new features and improvements

We added the following new features:

DataFrame:

- `last_valid_index` (#1705)

Series:

- `product` (#1677)
- `last_valid_index` (#1705)

GroupBy:

- `cumcount` (#1702)

### 5.9.5 Other improvements

- Refine Spark I/O. (#1667)
- Set `partitionBy` explicitly in `to_parquet`.
- Add `mode` and `partition_cols` to `to_csv` and `to_json`.
- Fix type hints to use `Optional`.
- Make `read_excel` read from DFS if the underlying Spark is 3.0.0 or above. (#1678, #1693, #1694, #1692)
- Support callable instances to apply as a function, and fix `groupby.apply` to keep the index when possible (#1686)
- Bug fixing for `hasnans` when `non-DoubleType`. (#1681)
- Support `axis=1` for `DataFrame.dropna()`. (#1689)
- Allow assigning index as a column (#1696)
- Try to read pandas metadata in `read_parquet` if `index_col` is `None`. (#1695)
- Include pandas `Index` object in dataframe indexing options (#1698)
- Unified `PlotAccessor` for `DataFrame` and `Series` (#1662)

- Fix SeriesGroupBy.nsmallest/nlargest. (#1713)
- Fix DataFrame.size to consider its number of columns. (#1715)
- Fix first\_valid\_index() for Empty object (#1704)
- Fix index name when groupby.apply returns a single row. (#1719)
- Support subtraction of date/timestamp with literals. (#1721)
- DataFrame.reindex(fill\_value) does not fill existing NaN values (#1723)

## 5.10 Version 1.1.0

### 5.10.1 API extensions

We added support for API extensions (#1617).

You can register your custom accessors to DataFrame, Series, and Index.

For example, in your library code:

```
from databricks.koalas.extensions import register_dataframe_accessor

@register_dataframe_accessor("geo")
class GeoAccessor:

    def __init__(self, koalas_obj):
        self._obj = koalas_obj
        # other constructor logic

    @property
    def center(self):
        # return the geographic center point of this DataFrame
        lat = self._obj.latitude
        lon = self._obj.longitude
        return (float(lon.mean()), float(lat.mean()))

    def plot(self):
        # plot this array's data on a map
        pass

    ...
```

Then, in a session:

```
>>> from my_ext_lib import GeoAccessor
>>> kdf = ks.DataFrame({"longitude": np.linspace(0,10),
...                     "latitude": np.linspace(0, 20)})
>>> kdf.geo.center
(5.0, 10.0)

>>> kdf.geo.plot()
...
```

See also: <https://koalas.readthedocs.io/en/latest/reference/extensions.html>

### 5.10.2 Plotting backend

We introduced `plotting.backend` configuration (#1639).

Plotly (>=4.8) or other libraries that pandas supports can be used as a plotting backend if they are installed in the environment.

```
>>> kdf = ks.DataFrame([[1, 2, 3, 4], [5, 6, 7, 8]], columns=["A", "B", "C", "D"])
>>> kdf.plot(title="Example Figure") # defaults to backend="matplotlib"
```

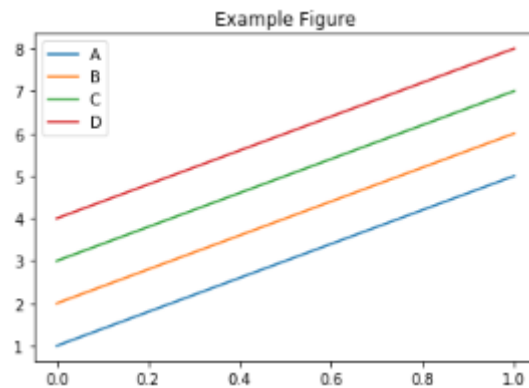


Fig. 4: image

```
>>> fig = kdf.plot(backend="plotly", title="Example Figure", height=500, width=500)
>>> ## same as:
>>> # ks.options.plotting.backend = "plotly"
>>> # fig = kdf.plot(title="Example Figure", height=500, width=500)
>>> fig.show()
```

Each backend returns the figure in their own format, allowing for further editing or customization if required.

```
>>> fig.update_layout(template="plotly_dark")
>>> fig.show()
```

### 5.10.3 Koalas accessor

We introduced `koalas` accessor and some methods specific to Koalas (#1613, #1628).

`DataFrame.apply_batch`, `DataFrame.transform_batch`, and `Series.transform_batch` are deprecated and moved to `koalas` accessor.

```
>>> kdf = ks.DataFrame({'a': [1,2,3], 'b': [4,5,6]})
>>> def pandas_plus(pdf):
...     return pdf + 1 # should always return the same length as input.
...
>>> kdf.koalas.transform_batch(pandas_plus)
   a  b
0  2  5
1  3  6
2  4  7
```

Example Figure

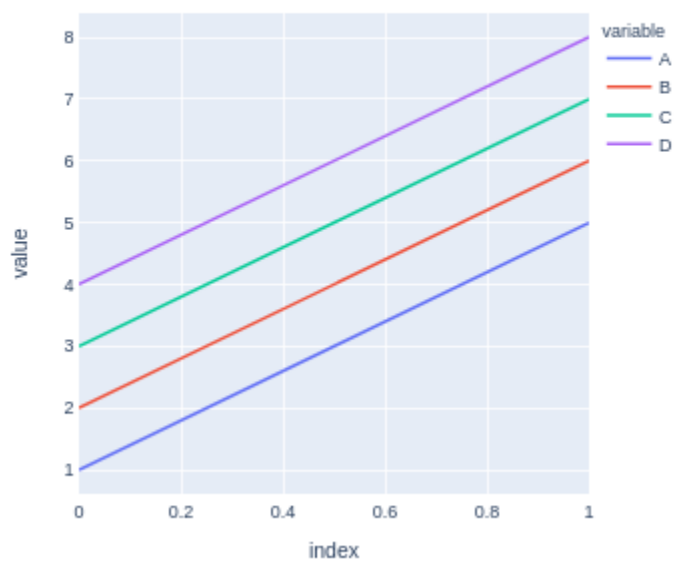


Fig. 5: image

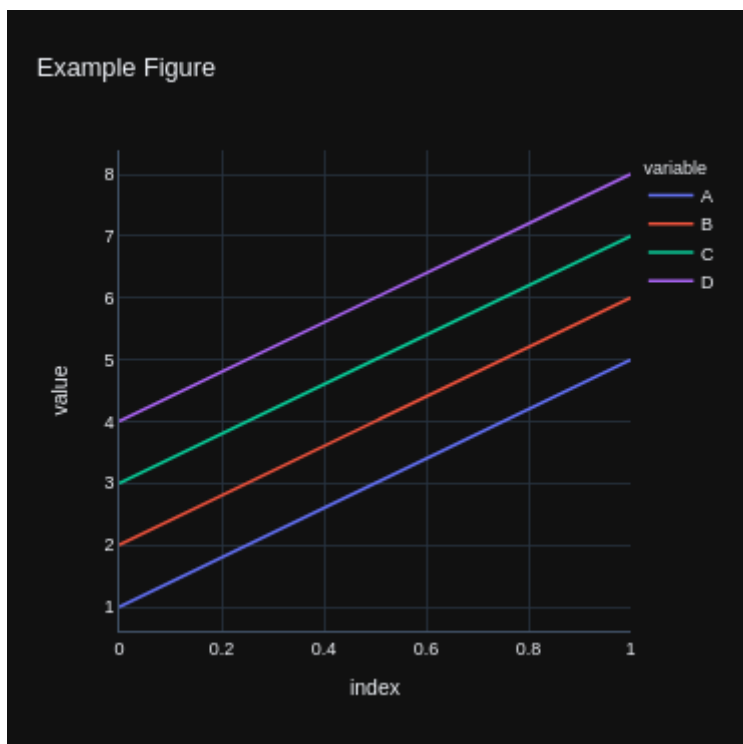


Fig. 6: image

```
>>> kdf = ks.DataFrame({'a': [1,2,3], 'b':[4,5,6]})
>>> def pandas_filter(pdf):
...     return pdf[pdf.a > 1] # allow arbitrary length
...
>>> kdf.koalas.apply_batch(pandas_filter)
   a  b
1  2  5
2  3  6
```

or

```
>>> kdf = ks.DataFrame({'a': [1,2,3], 'b':[4,5,6]})
>>> def pandas_plus(pser):
...     return pser + 1 # should always return the same length as input.
...
>>> kdf.a.koalas.transform_batch(pandas_plus)
0    2
1    3
2    4
Name: a, dtype: int64
```

See also: [https://koalas.readthedocs.io/en/latest/user\\_guide/transform\\_apply.html](https://koalas.readthedocs.io/en/latest/user_guide/transform_apply.html)

## 5.10.4 Other new features and improvements

We added the following new features:

DataFrame:

- `tail` (#1632)
- `droplevel` (#1622)

Series:

- `iteritems` (#1603)
- `items` (#1603)
- `tail` (#1632)
- `droplevel` (#1630)

## 5.10.5 Other improvements

- Simplify `Series.to_frame`. (#1624)
- Make Window functions create a new DataFrame. (#1623)
- Fix `Series._with_new_scol` to use alias. (#1634)
- Refine `concat` to handle the same anchor DataFrames properly. (#1627)
- Add `sort` parameter to `concat`. (#1636)
- Enable to assign list. (#1644)
- Use `SPARK_INDEX_NAME_FORMAT` in `combine_frames` to avoid ambiguity. (#1650)
- Rename spark columns only when `index=False`. (#1649)



- `read_csv`: Implement reading of number of rows (#1656)
- Fixed `ks.Index.to_series()` to work properly with name parameter (#1643)
- Fix `fillna` to handle “ffill” and “bfill” properly. (#1654)

## 5.11 Version 1.0.1

### 5.11.1 Critical bug fix

We fixed a critical bug introduced in Koalas 1.0.0 (#1609).

If we call `DataFrame.rename` with `columns` parameter after some operations on the `DataFrame`, the operations will be lost:

```
>>> kdf = ks.DataFrame([[1, 2, 3, 4], [5, 6, 7, 8]], columns=["A", "B", "C", "D"])
>>> kdf1 = kdf + 1
>>> kdf1
   A  B  C  D
0  2  3  4  5
1  6  7  8  9
>>> kdf1.rename(columns={"A": "aa", "B": "bb"})
   aa  bb  C  D
0   1   2  3  4
1   5   6  7  8
```

This should be:

```
>>> pdf1.rename(columns={"A": "aa", "B": "bb"})
   aa  bb  C  D
0   2   3  4  5
1   6   7  8  9
```

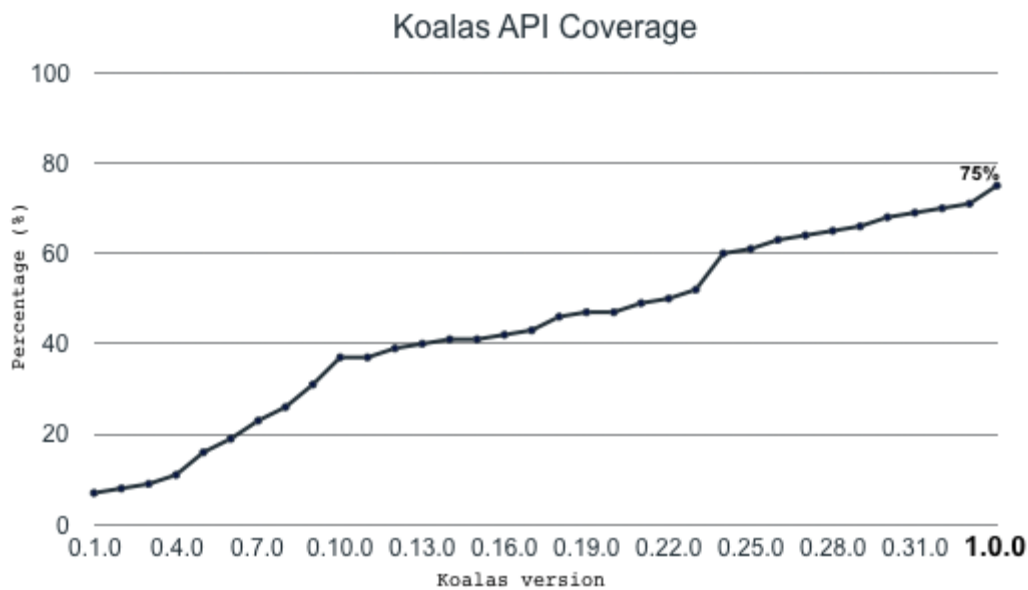
### 5.11.2 Other improvements

- Clean up `InternalFrame` and around anchor. (#1601)
- Fixing `DataFrame.iteritems` to return generator (#1602)
- Clean up `groupby` to use the anchor. (#1610)

## 5.12 Version 1.0.0

### 5.12.1 Better pandas API coverage

We implemented many APIs and features equivalent with pandas such as plotting, grouping, windowing, I/O, and transformation, and now Koalas reaches the pandas API coverage close to 80% in Koalas 1.0.0.



### 5.12.2 Apache Spark 3.0

Apache Spark 3.0 is now supported in Koalas 1.0 (#1586, #1558). Koalas does not require any change to use Spark 3.0. Apache Spark has [more than 3400 fixes landed in Spark 3.0](#) and Koalas shares the most of fixes in many other components.

It also brings the performance improvement in Koalas APIs that execute Python native functions internally via pandas UDFs, for example, `DataFrame.apply` and `DataFrame.apply_batch` (#1508).

### 5.12.3 Python 3.8

With Apache Spark 3.0, Koalas supports the latest Python 3.8 which has many significant improvements (#1587), see also [Python 3.8.0 release notes](#).

### 5.12.4 Spark accessor

`spark` accessor was introduced from Koalas 1.0.0 in order for the Koalas users to leverage the existing PySpark APIs more easily (#1530). For example, you can apply the PySpark functions as below:

```
import databricks.koalas as ks
import pyspark.sql.functions as F

kss = ks.Series([1, 2, 3, 4])
kss.spark.apply(lambda s: F.collect_list(s))
```

### 5.12.5 Better type hint support

In the early versions, it was required to use Koalas instances as the return type hints for the functions that return a pandas instances, which looks slightly awkward.

```
def pandas_div(pdf) -> koalas.DataFrame[float, float]:
    # pdf is a pandas DataFrame,
    return pdf[['B', 'C']] / pdf[['B', 'C']]

df = ks.DataFrame({'A': ['a', 'a', 'b'], 'B': [1, 2, 3], 'C': [4, 6, 5]})
df.groupby('A').apply(pandas_div)
```

In Koalas 1.0.0 with Python 3.7+, you can also use pandas instances in the return type as below:

```
def pandas_div(pdf) -> pandas.DataFrame[float, float]:
    return pdf[['B', 'C']] / pdf[['B', 'C']]
```

In addition, the new type hinting is experimentally introduced in order to allow users to specify column names in the type hints as below (#1577):

```
def pandas_div(pdf) -> pandas.DataFrame['B': float, 'C': float]:
    return pdf[['B', 'C']] / pdf[['B', 'C']]
```

See also [the guide](#) in Koalas documentation (#1584) for more details.

### 5.12.6 Wider support of in-place update

Previously in-place updates happen only within each DataFrame or Series, but now the behavior follows pandas in-place updates and the update of one side also updates the other side (#1592).

For example, the following updates kdf as well.

```
kdf = ks.DataFrame({"x": [np.nan, 2, 3, 4, np.nan, 6]})
kser = kdf.x
kser.fillna(0, inplace=True)
```

```
kdf = ks.DataFrame({"x": [np.nan, 2, 3, 4, np.nan, 6]})
kser = kdf.x
kser.loc[2] = 30
```

```
kdf = ks.DataFrame({"x": [np.nan, 2, 3, 4, np.nan, 6]})
kser = kdf.x
kdf.loc[2, 'x'] = 30
```

If the DataFrame and Series are connected, the in-place updates update each other.

### 5.12.7 Less restriction on `compute.ops_on_diff_frames`

In Koalas 1.0.0, the restriction of `compute.ops_on_diff_frames` became much more loosened (#1522, #1554). For example, the operations such as below can be performed without enabling `compute.ops_on_diff_frames`, which can be expensive due to the shuffle under the hood.

```
df + df + df
df['foo'] = df['bar']['baz']
df[['x', 'y']] = df[['x', 'y']].fillna(0)
```

### 5.12.8 Other new features and improvements

DataFrame:

- `__bool__` (#1526)
- `explode` (#1507)
- `spark.apply` (#1536)
- `spark.schema` (#1530)
- `spark.print_schema` (#1530)
- `spark.frame` (#1530)
- `spark.cache` (#1530)
- `spark.persist` (#1530)
- `spark.hint` (#1530)
- `spark.to_table` (#1530)
- `spark.to_spark_io` (#1530)
- `spark.explain` (#1530)
- `spark.apply` (#1530)
- `mad` (#1538)
- `__abs__` (#1561)

Series:

- `item` (#1502, #1518)
- `divmod` (#1397)
- `rdivmod` (#1397)
- `unstack` (#1501)
- `mad` (#1503)
- `__bool__` (#1526)
- `to_markdown` (#1510)
- `spark.apply` (#1536)
- `spark.data_type` (#1530)
- `spark.nullable` (#1530)

- `spark.column` (#1530)
- `spark.transform` (#1530)
- `filter` (#1511)
- `__abs__` (#1561)
- `bfill` (#1580)
- `ffill` (#1580)

Index:

- `__bool__` (#1526)
- `spark.data_type` (#1530)
- `spark.column` (#1530)
- `spark.transform` (#1530)
- `get_level_values` (#1517)
- `delete` (#1165)
- `__abs__` (#1561)
- `holds_integer` (#1547)

MultiIndex:

- `__bool__` (#1526)
- `spark.data_type` (#1530)
- `spark.column` (#1530)
- `spark.transform` (#1530)
- `get_level_values` (#1517)
- `delete` (#1165)
- `__abs__` (#1561)
- `holds_integer` (#1547)

Along with the following improvements:

- Fix `Series.clip` not to create a new `DataFrame`. (#1525)
- Fix `combine_first` to support tupled names. (#1534)
- Add Spark accessors to usage logging. (#1540)
- Implements multi-index support in `Dataframe.filter` (#1512)
- Fix `Series.fillna` to avoid Spark jobs. (#1550)
- Support `DataFrame.spark.explain(extended: str)` case. (#1563)
- Support Series as repeats in `Series.repeat`. (#1573)
- Fix `fillna` to handle NaN properly. (#1572)
- Fix `DataFrame.replace` to avoid creating a new Spark `DataFrame`. (#1575)
- Cache an internal pandas object to avoid run twice in Jupyter. (#1564)
- Fix `Series.div` when `div/floordiv np.inf` by zero (#1463)

- Fix Series.unstack to support non-numeric type and keep the names (#1527)
- Fix hasnans to follow the modified column. (#1532)
- Fix explode to use internal methods. (#1538)
- Fix RollingGroupby and ExpandingGroupby to handle agg\_columns. (#1546)
- Fix reindex not to update internal. (#1582)

### 5.12.9 Backward Compatibility

- Remove the deprecated pandas\_wraps (#1529)
- Remove compute function. (#1531)

## 5.13 Version 0.33.0

### 5.13.1 apply and transform Improvements

We added supports to have positional/keyword arguments for apply, apply\_batch, transform, and transform\_batch in DataFrame, Series, and GroupBy. (#1484, #1485, #1486)

```
>>> ks.range(10).apply(lambda a, b, c: a + b + c, args=(1,), c=3)
   id
0    4
1    5
2    6
3    7
4    8
5    9
6   10
7   11
8   12
9   13
```

```
>>> ks.range(10).transform_batch(lambda pdf, a, b, c: pdf.id + a + b + c, 1, 2, c=3)
0      6
1      7
2      8
3      9
4     10
5     11
6     12
7     13
8     14
9     15
Name: id, dtype: int64
```

```
>>> kdf = ks.DataFrame(
...     {"a": [1, 2, 3, 4, 5, 6], "b": [1, 1, 2, 3, 5, 8], "c": [1, 4, 9, 16, 25, 36]},
...     columns=["a", "b", "c"])
>>> kdf.groupby(["a", "b"]).apply(lambda x, y, z: x + x.min() + y + z, 1, z=2)
   a  b  c
0  5  5  5
```

(continues on next page)

(continued from previous page)

1	7	5	11
2	9	7	21
3	11	9	35
4	13	13	53
5	15	19	75

### 5.13.2 Spark Schema

We add `spark_schema` and `print_schema` to know the underlying Spark Schema. (#1446)

```
>>> kdf = ks.DataFrame({'a': list('abc'),
...                     'b': list(range(1, 4)),
...                     'c': np.arange(3, 6).astype('i1'),
...                     'd': np.arange(4.0, 7.0, dtype='float64'),
...                     'e': [True, False, True],
...                     'f': pd.date_range('20130101', periods=3)},
...                     columns=['a', 'b', 'c', 'd', 'e', 'f'])

>>> # Print the schema out in Spark's DDL formatted string
>>> kdf.spark_schema().simpleString()
'struct<a:string,b:bigint,c:tinyint,d:double,e:boolean,f:timestamp>'
>>> kdf.spark_schema(index_col='index').simpleString()
'struct<index:bigint,a:string,b:bigint,c:tinyint,d:double,e:boolean,f:timestamp>'

>>> # Print out the schema as same as DataFrame.printSchema()
>>> kdf.print_schema()
root
 |-- a: string (nullable = false)
 |-- b: long (nullable = false)
 |-- c: byte (nullable = false)
 |-- d: double (nullable = false)
 |-- e: boolean (nullable = false)
 |-- f: timestamp (nullable = false)

>>> kdf.print_schema(index_col='index')
root
 |-- index: long (nullable = false)
 |-- a: string (nullable = false)
 |-- b: long (nullable = false)
 |-- c: byte (nullable = false)
 |-- d: double (nullable = false)
 |-- e: boolean (nullable = false)
 |-- f: timestamp (nullable = false)
```

### 5.13.3 GroupBy Improvements

We fixed many bugs of `GroupBy` as listed below.

- Fix `groupby` when `as_index=False`. (#1457)
- Make `groupby.apply` in `pandas<0.25` run the function only once per group. (#1462)
- Fix `Series.groupby` on the `Series` from different `DataFrames`. (#1460)
- Fix `GroupBy.head` to recognize `agg_columns`. (#1474)
- Fix `GroupBy.filter` to follow complex group keys. (#1471)
- Fix `GroupBy.transform` to follow complex group keys. (#1472)
- Fix `GroupBy.apply` to follow complex group keys. (#1473)
- Fix `GroupBy.fillna` to use `GroupBy._apply_series_op`. (#1481)
- Fix `GroupBy.filter` and `apply` to handle `agg_columns`. (#1480)
- Fix `GroupBy.apply`, `filter`, and `head` to ignore temp columns when ops from different `DataFrames`. (#1488)
- Fix `GroupBy` functions which need natural orderings to follow the order when ops from different `DataFrames`. (#1490)

### 5.13.4 Other new features and improvements

We added the following new feature:

`SeriesGroupBy`:

- `filter` (#1483)

### 5.13.5 Other improvements


- `dtype` for `DateType` should be `np.dtype("object")`. (#1447)
- Make `reset_index` disallow the same name but allow it when `drop=True`. (#1455)
- Fix named aggregation for `MultiIndex` (#1435)
- Raise `ValueError` that is not raised now (#1461)
- Fix `get_dummies` when uses the `prefix` parameter whose type is `dict` (#1478)
- Simplify `DataFrame.columns` setter. (#1489)

## 5.14 Version 0.32.0

### 5.14.1 Koalas documentation redesign

Koalas documentation was redesigned with a better theme, `pydata-sphinx-theme`. Please check the new [Koalas documentation site](#) out.




[Getting started](#)
[User Guide](#)
[API Reference](#)
[Development](#)
[Release Notes](#)

## Installation

Koalas requires PySpark so please make sure your PySpark is available.

To install Koalas, you can use:

- [Conda](#)
- [PyPI](#)
- [Installation from source](#)

On this page

- Python version support
- Installing Koalas
- Installing PySpark
- Dependencies

**Installation**

10 minutes to Koalas

Koalas Talks and Blogs

### 5.14.2 transform\_batch and apply\_batch

We added the APIs that enable you to directly transform and apply a function against Koalas Series or DataFrame. `map_in_pandas` is deprecated and now renamed to `apply_batch`.

```
import databricks.koalas as ks
kdf = ks.DataFrame({'a': [1,2,3], 'b': [4,5,6]})
def pandas_plus(pdf):
    return pdf + 1 # should always return the same length as input.

kdf.transform_batch(pandas_plus)
```

```
import databricks.koalas as ks
kdf = ks.DataFrame({'a': [1,2,3], 'b': [4,5,6]})
def pandas_plus(pdf):
    return pdf[pdf.a > 1] # allow arbitrary length

kdf.apply_batch(pandas_plus)
```

Please also check [Transform and apply a function](#) in Koalas documentation.

### 5.14.3 Other new features and improvements

We added the following new feature:

DataFrame:

- `truncate` (#1408)
- `hint` (#1415)

SeriesGroupBy:

- `unique` (#1426)

Index:

- `spark_column` (#1438)

Series:

- `spark_column` (#1438)

MultiIndex:

- `spark_column` (#1438)

### 5.14.4 Other improvements

- Fix from\_pandas to handle the same index name as a column name. (#1419)
- Add documentation about non-Koalas APIs (#1420)
- Hot-fixing the lack of keyword argument ‘deep’ for DataFrame.copy() (#1423)
- Fix Series.div when divide by zero (#1412)
- Support expand parameter if n is a positive integer in Series.str.split/rsplit. (#1432)
- Make Series.astype(bool) follow the concept of “truthy” and “falsey”. (#1431)
- Fix incompatible behaviour with pandas for floordiv with np.nan (#1429)
- Use mapInPandas for apply\_batch API in Spark 3.0 (#1440)
- Use F.datediff() for subtraction of dates as a workaround. (#1439)

## 5.15 Version 0.31.0

### 5.15.1 PyArrow>=0.15 support is back

We added PyArrow>=0.15 support back (#1110).

Note that, when working with pyarrow>=0.15 and pyspark<3.0, Koalas will set an environment variable ARROW\_PRE\_0\_15\_IPC\_FORMAT=1 if it does not exist, as per the instruction in [SPARK-29367](#), but it will NOT work if there is a Spark context already launched. In that case, you have to manage the environment variable by yourselves.

### 5.15.2 Spark specific improvements

#### Broadcast hint

We added broadcast function in namespace.py (#1360).

We can use it with merge, join, and update which invoke join operation in Spark when you know one of the DataFrame is small enough to fit in memory, and we can expect much more performant than shuffle-based joins.

For example,

```
>>> merged = df1.merge(ks.broadcast(df2), left_on='lkey', right_on='rkey')
>>> merged.explain()
== Physical Plan ==
...
...BroadcastHashJoin...
...
```

## persist function and storage level

We added `persist` function to specify the storage level when caching (#1381), and also, we added `storage_level` property to check the current storage level (#1385).

```
>>> with df.cache() as cached_df:
...     print(cached_df.storage_level)
...
Disk Memory Deserialized 1x Replicated

>>> with df.persist(pyspark.StorageLevel.MEMORY_ONLY) as cached_df:
...     print(cached_df.storage_level)
...
Memory Serialized 1x Replicated
```

## 5.15.3 Other new features and improvements

We added the following new feature:

DataFrame:

- `to_markdown` (#1377)
- `squeeze` (#1389)

Series:

- `squeeze` (#1389)
- `asof` (#1366)

## 5.15.4 Other improvements

- Add a way to specify index column in I/O APIs (#1379)
- Fix `iloc.__setitem__` with the other Series from the same DataFrame. (#1388)
- Add support Series from different DataFrames for `loc/iloc.__setitem__`. (#1391)
- Refine `__setitem__` for `loc/iloc` with DataFrame. (#1394)
- Help misuse of `options` argument. (#1402)
- Add blog posts in Koalas documentation (#1406)
- Fix `mod` & `rmod` for matching with pandas. (#1399)

## 5.16 Version 0.30.0

### 5.16.1 Slice column selection support in `loc`

We continue to improve `loc` indexer and added the slice column selection support (#1351).

```
>>> from databricks import koalas as ks
>>> df = ks.DataFrame({'a':list('abcdefghij'), 'b':list('abcdefghij'), 'c': range(10)})
↪
>>> df.loc[:, "b":"c"]
   b  c
0  a  0
1  b  1
2  c  2
3  d  3
4  e  4
5  f  5
6  g  6
7  h  7
8  i  8
9  j  9
```

### 5.16.2 Slice row selection support in loc for multi-index

We also added the support of slice as row selection in `loc` indexer for multi-index (#1344).

```
>>> from databricks import koalas as ks
>>> import pandas as pd
>>> df = ks.DataFrame({'a': range(3)}, index=pd.MultiIndex.from_tuples([("a", "b"), (
↪ "a", "c"), ("b", "d")]))
>>> df.loc[("a", "c"): "b"]
      a
a c  1
b d  2
```

### 5.16.3 Slice row selection support in iloc

We continued to improve `iloc` indexer to support iterable indexes as row selection (#1338).

```
>>> from databricks import koalas as ks
>>> df = ks.DataFrame({'a':list('abcdefghij'), 'b':list('abcdefghij')})
>>> df.iloc[[-1, 1, 2, 3]]
   a  b
1  b  b
2  c  c
3  d  d
9  j  j
```

### 5.16.4 Support of setting values via loc and iloc at Series

Now, we added the basic support of setting values via `loc` and `iloc` at Series (#1367).

```
>>> from databricks import koalas as ks
>>> kser = ks.Series([1, 2, 3], index=["cobra", "viper", "sidewinder"])
>>> kser.loc[kser % 2 == 1] = -kser
>>> kser
cobra      -1
viper       2
sidewinder -3
```

### 5.16.5 Other new features and improvements

We added the following new feature:

DataFrame:

- `take` (#1292)
- `eval` (#1359)

Series:

- `dot` (#1136)
- `take` (#1357)
- `combine_first` (#1290)

Index:

- `droplevel` (#1340)
- `union` (#1348)
- `take` (#1357)
- `asof` (#1350)

MultiIndex:

- `droplevel` (#1340)
- `unique` (#1342)
- `union` (#1348)
- `take` (#1357)

### 5.16.6 Other improvements

- Compute `Index.is_monotonic/Index.is_monotonic_decreasing` in a distributed manner (#1354)
- Fix `SeriesGroupBy.apply()` to respect various output (#1339)
- Add the support for operations between different DataFrames in `groupby()` (#1321)
- Explicitly don't support to disable `numeric_only` in stats APIs at DataFrame (#1343)
- Fix index operator against Series and Frame to use `iloc` conditionally (#1336)
- Make `nunique` in DataFrame to return a Koalas DataFrame instead of pandas' (#1347)
- Fix `MultiIndex.drop()` to follow renaming et al. (#1356)
- Add column axis in `ks.concat` (#1349)
- Fix `iloc` for Series when the series is modified. (#1368)
- Support MultiIndex for duplicated, `drop_duplicates`. (#1363)

## 5.17 Version 0.29.0

### 5.17.1 Slice support in `iloc`

We improved `iloc` indexer to support slice as row selection. (#1335)

For example,

```
>>> kdf = ks.DataFrame({'a':list('abcdefghij')})
>>> kdf
   a
0  a
1  b
2  c
3  d
4  e
5  f
6  g
7  h
8  i
9  j
>>> kdf.iloc[2:5]
   a
2  c
3  d
4  e
>>> kdf.iloc[2:-3:2]
   a
2  c
4  e
6  g
>>> kdf.iloc[5:]
   a
5  f
6  g
7  h
8  i
9  j
>>> kdf.iloc[5:2]
Empty DataFrame
Columns: [a]
Index: []
```

### 5.17.2 Documentation

We added links to the previous talks in our document. (#1319)

You can see a lot of useful talks from the previous events and we will keep updated.

[https://koalas.readthedocs.io/en/latest/getting\\_started/videos.html](https://koalas.readthedocs.io/en/latest/getting_started/videos.html)

### 5.17.3 Other new features and improvements

We added the following new feature:

DataFrame: - `stack` (#1329)

Series:

- `repeat` (#1328)

Index:

- `difference` (#1325)
- `repeat` (#1328)

MultiIndex:

- `difference` (#1325)
- `repeat` (#1328)

### 5.17.4 Other improvements

- `DataFrame.pivot` should preserve the original index names. (#1316)
- Fix `_LocIndexerLike` to handle a Series from index. (#1315)
- Support MultiIndex in `DataFrame.unstack`. (#1322)
- Support Spark UDT when converting from/to pandas DataFrame/Series. (#1324)
- Allow negative numbers for `head`. (#1330)
- Return a Koalas series instead of pandas' in stats APIs at Koalas DataFrame (#1333)

## 5.18 Version 0.28.0

### 5.18.1 pandas 1.0 support

We added pandas 1.0 support (#1197, #1299), and Koalas now can work with pandas 1.0.

### 5.18.2 map\_in\_pandas

We implemented `DataFrame.map_in_pandas` API (#1276) so Koalas can allow any arbitrary function with pandas DataFrame against Koalas DataFrame. See the example below:

```
>>> import databricks.koalas as ks
>>> df = ks.DataFrame({'A': range(2000), 'B': range(2000)})
>>> def query_func(pdf):
...     num = 1995
...     return pdf.query('A > @num')
...
>>> df.map_in_pandas(query_func)
      A      B
1996  1996  1996
1997  1997  1997
```

(continues on next page)

(continued from previous page)

1998	1998	1998
1999	1999	1999

### 5.18.3 Standardize code style using Black

As a development only change, we added [Black](#) integration ([#1301](#)). Now, all code style is standardized automatically via running `./dev/reformat`, and the style is checked as a part of `./dev/lint-python`.

### 5.18.4 Other new features and improvements

We added the following new feature:

DataFrame:

- `query` ([#1273](#))
- `unstack` ([#1295](#))

### 5.18.5 Other improvements

- Fix `DataFrame.describe()` to support multi-index columns. ([#1279](#))
- Add util function `validate_bool_kwarg` ([#1281](#))
- Rename data columns prior to filter to make sure the column names are as expected. ([#1283](#))
- Add an `faq` about Structured Streaming. ([#1298](#))
- Let extra options have higher priority to allow workarounds ([#1296](#))
- Implement `'keep'` parameter for `drop_duplicates` ([#1303](#))
- Add a note when type hint is provided to `DataFrame.apply` ([#1310](#))
- Add a util method to verify temporary column names. ([#1262](#))

## 5.19 Version 0.27.0

### 5.19.1 head ordering

Since Koalas doesn't guarantee the row ordering, `head` could return some rows from distributed partition and the result is not deterministic, which might confuse users.

We added a configuration `compute.ordered_head` ([#1231](#)), and if it is set to `True`, Koalas performs natural ordering beforehand and the result will be the same as pandas'. The default value is `False` because the ordering will cause a performance overhead.

```
>>> kdf = ks.DataFrame({'a': range(10)})
>>> pdf = kdf.to_pandas()
>>> pdf.head(3)
   a
0  0
1  1
```

(continues on next page)



(continued from previous page)

```

2  2

>>> kdf.head(3)
   a
5  5
6  6
7  7
>>> kdf.head(3)
   a
0  0
1  1
2  2

>>> ks.options.compute.ordered_head = True
>>> kdf.head(3)
   a
0  0
1  1
2  2
>>> kdf.head(3)
   a
0  0
1  1
2  2

```

### 5.19.2 GitHub Actions

We started trying to use GitHub Actions for CI. (#1254, #1265, #1264, #1267, #1269)

### 5.19.3 Other new features and improvements

We added the following new feature:

DataFrame: - apply (#1259)

### 5.19.4 Other improvements

- Fix identical and equals for the comparison between the same object. (#1220)
- Select the series correctly in SeriesGroupBy APIs (#1224)
- Fixes DataFrame/Series.clip function to preserve its index. (#1232)
- Throw a better exception in DataFrame.sort\_values when multi-index column is used (#1238)
- Fix fillna not to change index values. (#1241)
- Fix DataFrame.\_\_setitem\_\_ with tuple-named Series. (#1245)
- Fix corr to support multi-index columns. (#1246)
- Fix output of print() matches with pandas of Series (#1250)
- Fix fillna to support partial column index for multi-index columns. (#1244)
- Add as\_index check logic to groupby parameter (#1253)

- Raising `NotImplementedError` for elements that actually are not implemented. (#1256)
- Fix where to support multi-index columns. (#1249)

## 5.20 Version 0.26.0

### 5.20.1 iat indexer

We continued to improve indexers. Now, `iat` indexer is supported too (#1062).

```
>>> df = ks.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                    columns=['A', 'B', 'C'])
>>> df
   A  B  C
0  0  2  3
1  0  4  1
2 10 20 30

>>> df.iat[1, 2]
1
```

### 5.20.2 Other new features and improvements

We added the following new features:

`koalas.Index`

- `equals` (#1216)
- `identical` (#1215)
- `is_all_dates` (#1205)
- `append` (#1163)
- `to_frame` (#1187)

`koalas.MultiIndex`:

- `equals` (#1216)
- `identical` (#1215)
- `swaplevel` (#1105)
- `is_all_dates` (#1205)
- `is_monotonic_increasing` (#1183)
- `is_monotonic_decreasing` (#1183)
- `append` (#1163)
- `to_frame` (#1187)

`koalas.DataFrameGroupBy`

- `describe` (#1168)

### 5.20.3 Other improvements

- Change default write mode to overwrite to be consistent with pandas (#1209)
- Prepare Spark 3 (#1211, #1181)
- Fix `DataFrame.idxmin/idxmax`. (#1198)
- Fix `reset_index` with the default index is “distributed-sequence”. (#1193)
- Fix column name as a tuple in multi column index (#1191)
- Add favicon to doc (#1189)

## 5.21 Version 0.25.0

### 5.21.1 `loc` and `iloc` indexers improvement

We improved `loc` and `iloc` indexers. Now, `loc` can support scalar values as indexers (#1172).

```
>>> import databricks.koalas as ks
>>>
>>> df = ks.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=['cobra', 'viper', 'sidewinder'],
...                    columns=['max_speed', 'shield'])
>>> df.loc['sidewinder']
max_speed    7
shield       8
Name: sidewinder, dtype: int64
>>> df.loc['sidewinder', 'max_speed']
7
```

In addition, Series derived from a different Frame can be used as indexers (#1155).

```
>>> import databricks.koalas as ks
>>>
>>> ks.options.compute.ops_on_diff_frames = True
>>>
>>> df1 = ks.DataFrame({'A': [0, 1, 2, 3, 4], 'B': [100, 200, 300, 400, 500]},
...                    index=[20, 10, 30, 0, 50])
>>> df2 = ks.DataFrame({'A': [0, -1, -2, -3, -4], 'B': [-100, -200, -300, -400, -500]}
↪,
...                    index=[20, 10, 30, 0, 50])
>>> df1.A.loc[df2.A > -3].sort_index()
10    1
20    0
30    2
```

Lastly, now `loc` uses its natural order according to index identically with pandas’ when using the slice (#1159, #1174, #1179). See the example below.

```
>>> df = ks.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=['cobra', 'viper', 'sidewinder'],
...                    columns=['max_speed', 'shield'])
>>> df.loc['cobra': 'viper', 'max_speed']
cobra    1
```

(continues on next page)

(continued from previous page)

```
viper      4
Name: max_speed, dtype: int64
```

### 5.21.2 Other new features and improvements

We added the following new features:

koalas.Series:

- `get` (#1153)

koalas.Index

- `drop` (#1117)
- `len` (#1161)
- `set_names` (#1134)
- `argmin` (#1162)
- `argmax` (#1162)

koalas.MultiIndex:

- `from_product` (#1144)
- `drop` (#1117)
- `len` (#1161)
- `set_names` (#1134)

### 5.21.3 Other improvements

- Add support `from_pandas` for `Index/MultiIndex`. (#1170)
- Add a hidden column `__natural_order__`. (#1146)
- Introduce `_LocIndexerLike` and consolidate some logic. (#1149)
- Refactor `LocIndexerLike.__getitem__`. (#1152)
- Remove `sort` in `GroupBy._reduce_for_stat_function`. (#1147)
- Randomize index in tests and fix some window-like functions. (#1151)
- Explicitly don't support `Index.duplicated` (#1131)
- Fix `DataFrame._repr_html_()`. (#1177)

## 5.22 Version 0.24.0

### 5.22.1 NumPy's universal function (ufunc) compatibility

We added the compatibility of NumPy ufunc ([#1127](#)). Virtually all ufunc compatibilities in Koalas DataFrame were implemented. See the example below:

```
>>> import databricks.koalas as ks
>>> import numpy as np
>>> kdf = ks.range(10)
>>> np.log(kdf)
      id
0      NaN
1  0.000000
2  0.693147
3  1.098612
4  1.386294
5  1.609438
6  1.791759
7  1.945910
8  2.079442
9  2.197225
```

### 5.22.2 Other new features and improvements

We added the following new features:

koalas:

- `to_numeric` ([#1060](#))

koalas.DataFrame:

- `idxmax` ([#1054](#))
- `idxmin` ([#1054](#))
- `pct_change` ([#1051](#))
- `info` ([#1124](#))

koalas.Index

- `fillna` ([#1102](#))
- `min` ([#1114](#))
- `max` ([#1114](#))
- `drop_duplicates` ([#1121](#))
- `nunique` ([#1132](#))
- `sort_values` ([#1120](#))

koalas.MultiIndex:

- `levshape` ([#1086](#))
- `min` ([#1114](#))
- `max` ([#1114](#))

- `sort_values` (#1120)

`koalas.SeriesGroupBy`

- `head` (#1050)

`koalas.DataFrameGroupBy`

- `head` (#1050)

### 5.22.3 Other improvements

- Setting index name / names for Series (#1079)
- disable 'str' for 'SeriesGroupBy', disable 'DataFrame' for 'GroupBy' (#1097)
- Support 'compute.ops\_on\_diff\_frames' for NumPy ufunc compay in Series (#1128)
- Support arithmetic and comparison APIs on same DataFrames (#1129)
- Fix `rename()` for Index to support MultiIndex also (#1125)
- Set the upper-bound for pandas. (#1137)
- Fix `_cum()` for Series to work properly (#1113)
- Fix `value_counts()` to work properly when `dropna` is True (#1116, #1142)

## 5.23 Version 0.23.0

### 5.23.1 NumPy's universal function (ufunc) compatibility

We added the compatibility of NumPy ufunc (#1096, #1106). Virtually all ufunc compatibilities in Koalas Series were implemented. See the example below:

```
>>> import databricks.koalas as ks
>>> import numpy as np
>>> kdf = ks.range(10)
>>> kser = np.sqrt(kdf.id)
>>> type(kser)
<class 'databricks.koalas.series.Series'>
>>> kser
0    0.000000
1    1.000000
2    1.414214
3    1.732051
4    2.000000
5    2.236068
6    2.449490
7    2.645751
8    2.828427
9    3.000000
```

### 5.23.2 Other new features and improvements

We added the following new features:

koalas:

- `option_context` (#1077)

`koalas.DataFrame`:

- `where` (#1018)
- `mask` (#1018)
- `iterrows` (#1070)

`koalas.Series`:

- `pop` (#866)
- `first_valid_index` (#1092)
- `pct_change` (#1071)

`koalas.Index`

- `symmetric_difference` (#953, #1059)
- `to_numpy` (#1058)
- `transpose` (#1056)
- `T` (#1056)
- `dropna` (#938)
- `shape` (#1085)
- `value_counts` (#949)

`koalas.MultiIndex`:

- `symmetric_difference` (#953, #1059)
- `to_numpy` (#1058)
- `transpose` (#1056)
- `T` (#1056)
- `dropna` (#938)
- `shape` (#1085)
- `value_counts` (#949)

### 5.23.3 Other improvements

- Fix comparison operators to treat NULL as False (#1029)
- Make `corr` return `koalas.DataFrame` (#1069)
- Include link to Help Thirsty Koalas Fund (#1082)
- Add Null handling for different frames (#1083)
- Allow `Series.__getitem__` to take boolean Series (#1075)

- Produce correct output against multiIndex when ‘compute.ops\_on\_diff\_frames’ is enabled (#1089)
- Fix idxmax() / idxmin() for Series work properly (#1078)

## 5.24 Version 0.22.0

### 5.24.1 Enable Arrow 0.15.1+

Apache Arrow 0.15.0 did not work well with PySpark 2.4 so it was disabled in the previous version. With Arrow 0.15.1, now it works in Koalas (#902).

### 5.24.2 Expanding and Rolling

We also added `expanding()` and `rolling()` APIs in all `groupby()`, `Series` and `Frame` (#985, #991, #990, #1015, #996, #1034, #1037)

- `min`
- `max`
- `sum`
- `mean`
- `std`
- `var`

### 5.24.3 Multi-index columns support

We continue improving multi-index columns support. We made the following APIs support multi-index columns:

- `median` (#995)
- `at` (#1049)

### 5.24.4 Documentation

We added “Best Practices” section in the documentation (#1041) so that Koalas users can read and follow. Please see [https://koalas.readthedocs.io/en/latest/user\\_guide/best\\_practices.html](https://koalas.readthedocs.io/en/latest/user_guide/best_practices.html)

### 5.24.5 Other new features and improvements

We added the following new features:

`koalas.DataFrame`:

- `quantile` (#984)
- `explain` (#1042)

`koalas.Series`:

- `between` (#997)
- `update` (#923)



- `mask` (#1017)

`koalas.MultiIndex`:

- `from_tuples` (#970)
- `from_arrays` (#1001)

Along with the following improvements:

- Introduce `column_scols` in `InternalFrame` substitute for `data_columns`. (#956)
- Fix different index level assignment when `'compute.ops_on_diff_frames'` is enabled (#1045)
- Fix `Dataframe.melt` function & Add doctest case for `melt` function (#987)
- Enable creating `Index` from list like `'Index([1, 2, 3])'` (#986)
- Fix `combine_frames` to handle where the right hand side arguments are modified `Series` (#1020)
- `setup.py` should support Python 2 to show a proper error message. (#1027)
- Remove `Series.schema`. (#993)

## 5.25 Version 0.21.0

### 5.25.1 Multi-index columns support

We continue improving multi-index columns support. We made the following APIs support multi-index columns:

- `nunique` (#980)
- `to_csv` (#983)

### 5.25.2 Documentation

Now, we have installation guide, design principles and FAQ in our public documentation (#914, #944, #963, #964)

### 5.25.3 Other new features and improvements

We added the following new features:

`koalas`

- `merge` (#969)

`koalas.DataFrame`:

- `keys` (#937)
- `ndim` (#947)

`koalas.Series`:

- `keys` (#935)
- `mode` (#899)
- `truncate` (#928)
- `xs` (#921)

- `where` (#922)
- `first_valid_index` (#936)

`koalas.Index`:

- `copy` (#939)
- `unique` (#912)
- `ndim` (#947)
- `has_duplicates` (#946)
- `nlevels` (#945)

`koalas.MultiIndex`:

- `copy` (#939)
- `ndim` (#947)
- `has_duplicates` (#946)
- `nlevels` (#945)

`koalas.Expanding`

- `count` (#978)

Along with the following improvements:

- Fix passing options as keyword arguments (#968)
- Make `is_monotonic` work properly for index (#930)
- Fix `Series.__getitem__` to work properly (#934)
- Fix `reindex` when all the given columns are included the existing columns (#975)
- Add `datetime` as the equivalent python type to `TimestampType` (#957)
- Fix `is_unique` to respect the current Spark column (#981)
- Fix bug when assign `None` to `name` as `Index` (#974)
- Use `name_like_string` instead of `str` directly. (#942, #950)

## 5.26 Version 0.20.0

### 5.26.1 Disable Arrow 0.15

Apache Arrow 0.15.0 was released on the 5th of October, 2019, which Koalas depends on to execute Pandas UDF, but the Spark community reports [an issue](#) with PyArrow 0.15.

We decided to set an upper bound for pyarrow version to avoid such issues until we are sure that Koalas works fine with it.

- Set an upper bound for pyarrow version. (#918)

### 5.26.2 Multi-index columns support

We continue improving multi-index columns support. We made the following APIs support multi-index columns:

- `pivot_table` (#908)
- `melt` (#920)

### 5.26.3 Other new features and improvements

We added the following new features:

`koalas.DataFrame`:

- `xs` (#892)

`koalas.Series`:

- `drop_duplicates` (#896)
- `replace` (#903)

`koalas.GroupBy`:

- `shift` (#910)

Along with the following improvements:

- Implement nested renaming for groupby agg (#904)
- Add 'index\_col' parameter to `DataFrame.to_spark` (#906)
- Add more options to `read_csv` (#916)
- Add `NamedAgg` (#911)
- Enable `DataFrame` setting value as list of labels (#905)

## 5.27 Version 0.19.0

### 5.27.1 Koalas Logo

Now that we have an official logo!



We can see the cute logo in our documents as well.

## 5.27.2 Documentation

Also we improved the documentation: <https://koalas.readthedocs.io/en/latest/>

- Added the logo (#831)
- Added a Jupyter notebook for 10 min tutorial (#843)
- Added the tutorial to the documentation (#853)
- Add some examples for plot implementations in their docstrings (#847)
- Move contribution guide to the official documentation site (#841)

### Binder integration for the 10 min tutorial

You can run a live Jupyter notebook for 10 min tutorial from .

## 5.27.3 Multi-index columns support

We continue improving multi-index columns support. We made the following APIs support multi-index columns:

- `transform` (#800)
- `round` (#802)
- `unique` (#809)
- `duplicated` (#803)
- `assign` (#811)
- `merge` (#825)
- `plot` (#830)
- `groupby` and its functions (#833)
- `update` (#848)
- `join` (#848)
- `drop_duplicate` (#856)
- `dtype` (#858)
- `filter` (#859)
- `dropna` (#857)
- `replace` (#860)

### 5.27.4 Plots

We also continue adding plot APIs as follows:

For DataFrame:

- `plot.kde()` (#784)

### 5.27.5 Other new features and improvements

We added the following new features:

`koalas.DataFrame`:

- `pop` (#791)
- `__iter__` (#836)
- `rename` (#806)
- `expanding` (#840)
- `rolling` (#840)

`koalas.Series`:

- `aggregate` (#816)
- `agg` (#816)
- `expanding` (#840)
- `rolling` (#840)
- `drop` (#829)
- `copy` (#869)

`koalas.DataFrameGroupBy`:

- `expanding` (#840)
- `rolling` (#840)

`koalas.SeriesGroupBy`:

- `expanding` (#840)
- `rolling` (#840)

Along with the following improvements:

- Add `squeeze` argument to `read_csv` (#812)
- Raise a more helpful error for duplicated columns in `Join` (#820)
- Issue with `ks.merge` to `Series` (#818)
- Fix `MultiIndex.to_pandas()` and `__repr__()`. (#832)
- Add `unit` and `origin` options for `to_datetime` (#839)
- Fix on wrong error raise in `DataFrame.fillna` (#844)
- Allow `str` and `list` in `aggfunc` in `DataFrameGroupby.agg` (#828)
- Add `index_col` argument to `to_koalas()`. (#863)

## 5.28 Version 0.18.0

### 5.28.1 Multi-index columns support

We continue improving multi-index columns support (#793, #776). We made the following APIs support multi-index columns:

- `applymap` (#793)
- `shift` (#793)
- `diff` (#793)
- `fillna` (#793)
- `rank` (#793)

Also, we can set tuple or None name for Series and Index. (#776)

```
>>> import databricks.koalas as ks
>>> kser = ks.Series([1, 2, 3])
>>> kser.name = ('a', 'b')
>>> kser
0    1
1    2
2    3
Name: (a, b), dtype: int64
```

### 5.28.2 Plots

We also continue adding plot APIs as follows:

For Series:

- `plot.kde()` (#767)

For DataFrame:

- `plot.hist()` (#780)

### 5.28.3 Options

In addition, we added the support for namespace-access in options (#785).

```
>>> import databricks.koalas as ks
>>> ks.options.display.max_rows
1000
>>> ks.options.display.max_rows = 10
>>> ks.options.display.max_rows
10
```

See also [User Guide](#) of our project docs.

### 5.28.4 Other new features and improvements

We added the following new features:

`koalas.DataFrame`:

- `aggregate` (#796)
- `agg` (#796)
- `items` (#787)

`koalas.indexes.Index/MultiIndex`

- `is_boolean` (#795)
- `is_categorical` (#795)
- `is_floating` (#795)
- `is_integer` (#795)
- `is_interval` (#795)
- `is_numeric` (#795)
- `is_object` (#795)

Along with the following improvements:

- Add `index_col` for `read_json` (#797)
- Add `index_col` for spark IO reads (#769, #775)
- Add “sep” parameter for `read_csv` (#777)
- Add axis parameter to `dataframe.diff` (#774)
- Add `read_json` and `to_json` use `spark.write.json` (#753)
- Use `spark.write.csv` in `to_csv` of `Series` and `DataFrame` (#749)
- Handle `TimestampType` separately when convert to pandas’ dtype. (#798)
- Fix `spark_df` when `set_index(..., drop=False)`. (#792)

### 5.28.5 Backward compatibility

- We removed some parameters in `DataFrame.to_csv` and `DataFrame.to_json` to allow distributed writing (#749, #753)

## 5.29 Version 0.17.0

### 5.29.1 Options

We started using options to configure the Koalas’ behavior. Now we have the following options:

- `display.max_rows` (#714, #742)
- `compute.max_rows` (#721, #736)
- `compute.shortcut_limit` (#717)
- `compute.ops_on_diff_frames` (#725)

- `compute.default_index_type` (#723)
- `plotting.max_rows` (#728)
- `plotting.sample_ratio` (#737)

We can also see the list and their descriptions in the [User Guide](#) of our project docs.

## 5.29.2 Plots

We continue adding plot APIs as follows:

For Series:

- `plot.area()` (#704)

For DataFrame:

- `plot.line()` (#686)
- `plot.bar()` (#695)
- `plot.barh()` (#698)
- `plot.pie()` (#703)
- `plot.area()` (#696)
- `plot.scatter()` (#719)

## 5.29.3 Multi-index columns support

We also continue improving multi-index columns support. We made the following APIs support multi-index columns:

- `koalas.concat()` (#680)
- `koalas.get_dummies()` (#695)
- `DataFrame.pivot_table()` (#635)

## 5.29.4 Other new features and improvements

We added the following new features:

`koalas`:

- `read_sql_table()` (#741)
- `read_sql_query()` (#741)
- `read_sql()` (#741)

`koalas.DataFrame`:

- `style` (#712)

Along with the following improvements:

- `GroupBy.apply` should return Koalas DataFrame instead of pandas DataFrame (#731)
- Fix `rpow` and `rfloordiv` to use proper operators in Series (#735)
- Fix `rpow` and `rfloordiv` to use proper operators in DataFrame (#740)



- Add schema inference support at `DataFrame.transform` (#732)
- Add `Option` class to support type check and value check in options (#739)
- Added missing tests (#687, #692, #694, #709, #711, #730, #729, #733, #734)

### 5.29.5 Backward compatibility

- We renamed two of the default index names from `one-by-one` and `distributed-one-by-one` to `sequence` and `distributed-sequence` respectively. (#679)
- We moved the configuration for enabling operations on different DataFrames from the environment variable to the option. (#725)
- We moved the configuration for the default index from the environment variable to the option. (#723)

## 5.30 Version 0.16.0

Firstly, we introduced new mode to enable operations on different DataFrames (#633). This mode can be enabled by setting `OPS_ON_DIFF_FRAMES` environment variable is set to `true` as below:

```
>>> import databricks.koalas as ks
>>>
>>> kdf1 = ks.range(5)
>>> kdf2 = ks.DataFrame({'id': [5, 4, 3]})
>>> (kdf1 - kdf2).sort_index()
      id
0 -5.0
1 -3.0
2 -1.0
3  NaN
4  NaN
```

```
>>> import databricks.koalas as ks
>>>
>>> kdf = ks.range(5)
>>> kdf['new_col'] = ks.Series([1, 2, 3, 4])
>>> kdf
      id  new_col
0     0         1.0
1     1         2.0
3     3         4.0
2     2         3.0
4     4         NaN
```

Secondly, we also introduced default index and disallowed Koalas DataFrame with no index internally (#639)(#655). For example, if you create Koalas DataFrame from Spark DataFrame, the default index is used. The default index implementation can be configured by setting `DEFAULT_INDEX` as one of three types:

- (default) `one-by-one`: It implements a one-by-one sequence by Window function without specifying partition. This index type should be avoided when the data is large.

```
>>> ks.range(3)
      id
0     0
```

(continues on next page)

(continued from previous page)

```
1  1
2  2
```

- `distributed-one-by-one`: It implements a one-by-one sequence by group-by and group-map approach. It still generates a one-by-one sequential index globally. If the default index must be a one-by-one sequence in a large dataset, this index can be used.

```
>>> ks.range(3)
      id
0      0
1      1
2      2
```

- `distributed`: It implements a monotonically increasing sequence simply by using Spark's `monotonically_increasing_id` function. If the index does not have to be a one-by-one sequence, this index can be used. Performance-wise, this index almost does not have any penalty comparing to other index types.

```
>>> ks.range(3)
      id
25769803776  0
60129542144  1
94489280512  2
```

Thirdly, we implemented many plot APIs in Series as follows:

- `plot.pie()` (#669)
- `plot.area()` (#670)
- `plot.line()` (#671)
- `plot.barh()` (#673)

See the example below:

```
import databricks.koalas as ks

ks.range(10).to_pandas().id.plot.pie()
```

Fourthly, we rapidly improved multi-index columns support continuously. Now multi-index columns are supported in multiple APIs:

- `DataFrame.sort_index()` (#637)
- `GroupBy.diff()` (#653)
- `GroupBy.rank()` (#653)
- `Series.any()` (#652)
- `Series.all()` (#652)
- `DataFrame.any()` (#652)
- `DataFrame.all()` (#652)
- `DataFrame.assign()` (#657)
- `DataFrame.drop()` (#658)
- `DataFrame.reindex()` (#659)

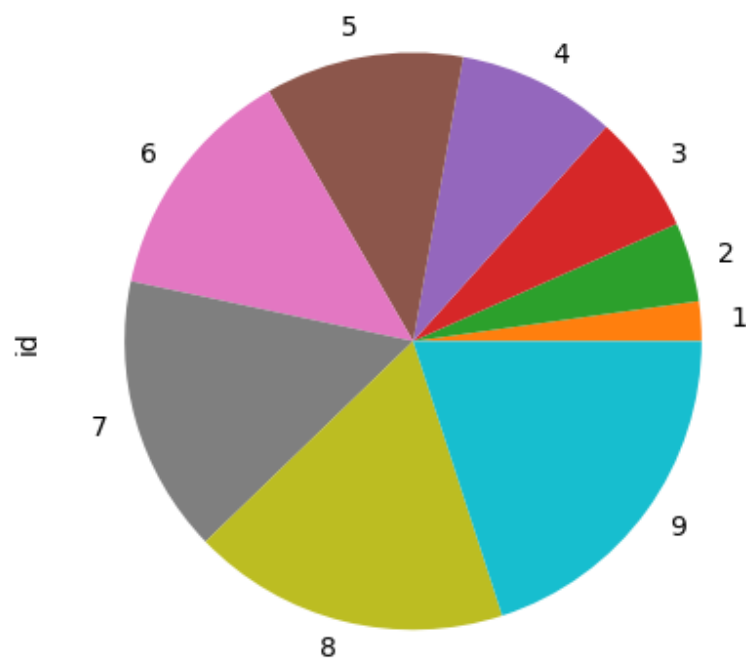


Fig. 7: image

- `Series.quantile()` (#663)
- `Series.transform()` (#663)
- `DataFrame.select_dtypes()` (#662)
- `DataFrame.transpose()` (#664).

Lastly we added new functionalities, especially for groupby-related functionalities, in the past weeks. We added the following features:

`koalas.DataFrame`

- `duplicated()` (#569)
- `fillna()` (#640)
- `bfill()` (#640)
- `pad()` (#640)
- `ffill()` (#640)

`koalas.groupby.GroupBy`:

- `diff()` (#622)
- `nunique()` (#617)
- `nlargest()` (#654)
- `nsmallest()` (#654)
- `idxmax()` (#649)
- `idxmin()` (#649)

Along with the following improvements:

- Add a basic infrastructure for configurations. (#645)
- Always use `column_index`. (#648)
- Allow to omit type hint in `GroupBy.transform`, `filter`, `apply` (#646)

## Symbols

`__init__()` (*databricks.koalas.CategoricalIndex method*), 670  
`__init__()` (*databricks.koalas.DataFrame method*), 342  
`__init__()` (*databricks.koalas.DatetimeIndex method*), 714  
`__init__()` (*databricks.koalas.Float64Index method*), 667  
`__init__()` (*databricks.koalas.Index method*), 612  
`__init__()` (*databricks.koalas.Int64Index method*), 664  
`__init__()` (*databricks.koalas.MultiIndex method*), 675  
`__init__()` (*databricks.koalas.Series method*), 96  
`__init__()` (*databricks.koalas.mlflow.PythonModelWrapper method*), 778

## A

`abs()` (*databricks.koalas.DataFrame method*), 461  
`abs()` (*databricks.koalas.Series method*), 155  
`add()` (*databricks.koalas.DataFrame method*), 386  
`add()` (*databricks.koalas.Series method*), 124  
`add_prefix()` (*databricks.koalas.DataFrame method*), 494  
`add_prefix()` (*databricks.koalas.Series method*), 206  
`add_suffix()` (*databricks.koalas.DataFrame method*), 494  
`add_suffix()` (*databricks.koalas.Series method*), 206  
`agg()` (*databricks.koalas.DataFrame method*), 451  
`agg()` (*databricks.koalas.groupby.DataFrameGroupBy method*), 747  
`agg()` (*databricks.koalas.Series method*), 146  
`aggregate()` (*databricks.koalas.DataFrame method*), 453  
`aggregate()` (*databricks.koalas.groupby.DataFrameGroupBy method*), 749  
`aggregate()` (*databricks.koalas.Series method*), 147  
`align()` (*databricks.koalas.DataFrame method*), 495  
`align()` (*databricks.koalas.Series method*), 200  
`all()` (*databricks.koalas.DataFrame method*), 462

`all()` (*databricks.koalas.groupby.GroupBy method*), 752  
`all()` (*databricks.koalas.Index method*), 627  
`all()` (*databricks.koalas.Series method*), 156  
`any()` (*databricks.koalas.DataFrame method*), 463  
`any()` (*databricks.koalas.groupby.GroupBy method*), 752  
`any()` (*databricks.koalas.Index method*), 627  
`any()` (*databricks.koalas.Series method*), 157  
`append()` (*databricks.koalas.DataFrame method*), 555  
`append()` (*databricks.koalas.Index method*), 658  
`append()` (*databricks.koalas.MultiIndex method*), 702  
`append()` (*databricks.koalas.Series method*), 246  
`apply()` (*databricks.koalas.DataFrame method*), 446  
`apply()` (*databricks.koalas.DataFrame.spark method*), 591  
`apply()` (*databricks.koalas.groupby.GroupBy method*), 742  
`apply()` (*databricks.koalas.Series method*), 145  
`apply()` (*databricks.koalas.Series.spark method*), 261  
`apply_batch()` (*databricks.koalas.DataFrame.koalas method*), 606  
`applymap()` (*databricks.koalas.DataFrame method*), 449  
`area()` (*databricks.koalas.DataFrame.plot method*), 595  
`area()` (*databricks.koalas.Series.plot method*), 316  
`argmax()` (*databricks.koalas.Index method*), 629  
`argmax()` (*databricks.koalas.Series method*), 237  
`argmin()` (*databricks.koalas.Index method*), 628  
`argmin()` (*databricks.koalas.Series method*), 237  
`argsort()` (*databricks.koalas.Series method*), 236  
`asof()` (*databricks.koalas.Index method*), 661  
`asof()` (*databricks.koalas.Series method*), 253  
`assign()` (*databricks.koalas.DataFrame method*), 556  
`astype()` (*databricks.koalas.DataFrame method*), 356  
`astype()` (*databricks.koalas.Index method*), 650  
`astype()` (*databricks.koalas.MultiIndex method*), 706  
`astype()` (*databricks.koalas.Series method*), 107  
`at()` (*databricks.koalas.DataFrame property*), 362  
`at()` (*databricks.koalas.Series property*), 109  
`at_time()` (*databricks.koalas.DataFrame method*),

497  
 at\_time() (*databricks.koalas.Series* method), 258  
 attach\_id\_column()  
     (*databricks.koalas.DataFrame.koalas* method),  
     605  
 axes() (*databricks.koalas.DataFrame* property), 351  
 axes() (*databricks.koalas.Series* property), 103

## B

backfill() (*databricks.koalas.DataFrame* method),  
     522  
 backfill() (*databricks.koalas.groupby.GroupBy*  
     method), 771  
 backfill() (*databricks.koalas.Series* method), 227  
 bar() (*databricks.koalas.DataFrame.plot* method), 597  
 bar() (*databricks.koalas.Series.plot* method), 317  
 barh() (*databricks.koalas.DataFrame.plot* method),  
     596  
 barh() (*databricks.koalas.Series.plot* method), 318  
 between() (*databricks.koalas.Series* method), 158  
 between\_time() (*databricks.koalas.DataFrame*  
     method), 498  
 between\_time() (*databricks.koalas.Series* method),  
     258  
 bfill() (*databricks.koalas.DataFrame* method), 528  
 bfill() (*databricks.koalas.groupby.GroupBy* method),  
     769  
 bfill() (*databricks.koalas.Series* method), 229  
 bool() (*databricks.koalas.DataFrame* method), 360  
 bool() (*databricks.koalas.Series* method), 108  
 box() (*databricks.koalas.Series.plot* method), 319  
 broadcast() (in module *databricks.koalas*), 84

## C

cache() (*databricks.koalas.DataFrame.spark* method),  
     585  
 capitalize() (*databricks.koalas.Series.str* method),  
     278  
 cat() (*databricks.koalas.Series.str* method), 279  
 CategoricalIndex (class in *databricks.koalas*), 669  
 categories() (*databricks.koalas.CategoricalIndex*  
     property), 674  
 categories() (*databricks.koalas.Series.cat* prop-  
     erty), 314  
 ceil() (*databricks.koalas.DatetimeIndex* method), 728  
 ceil() (*databricks.koalas.Series.dt* method), 275  
 center() (*databricks.koalas.Series.str* method), 279  
 checkpoint() (*databricks.koalas.DataFrame.spark*  
     method), 593  
 clip() (*databricks.koalas.DataFrame* method), 463  
 clip() (*databricks.koalas.Series* method), 159  
 coalesce() (*databricks.koalas.DataFrame.spark*  
     method), 592

codes() (*databricks.koalas.CategoricalIndex* prop-  
     erty), 673  
 codes() (*databricks.koalas.Series.cat* property), 315  
 column() (*databricks.koalas.Index.spark* property),  
     654  
 column() (*databricks.koalas.MultiIndex.spark* prop-  
     erty), 710  
 column() (*databricks.koalas.Series.spark* property),  
     260  
 columns() (*databricks.koalas.DataFrame* property),  
     349  
 combine\_first() (*databricks.koalas.Series*  
     method), 138  
 compare() (*databricks.koalas.Series* method), 247  
 concat() (in module *databricks.koalas*), 79  
 contains() (*databricks.koalas.Series.str* method),  
     280  
 copy() (*databricks.koalas.DataFrame* method), 355  
 copy() (*databricks.koalas.Index* method), 629  
 copy() (*databricks.koalas.MultiIndex* method), 693  
 copy() (*databricks.koalas.Series* method), 108  
 corr() (*databricks.koalas.DataFrame* method), 464  
 corr() (*databricks.koalas.Series* method), 159  
 count() (*databricks.koalas.DataFrame* method), 465  
 count() (*databricks.koalas.groupby.GroupBy* method),  
     753  
 count() (*databricks.koalas.Series* method), 160  
 count() (*databricks.koalas.Series.str* method), 281  
 count() (*databricks.koalas.window.Expanding*  
     method), 737  
 count() (*databricks.koalas.window.Rolling* method),  
     730  
 cumcount() (*databricks.koalas.groupby.GroupBy*  
     method), 754  
 cummax() (*databricks.koalas.DataFrame* method), 485  
 cummax() (*databricks.koalas.groupby.GroupBy*  
     method), 754  
 cummax() (*databricks.koalas.Series* method), 161  
 cummin() (*databricks.koalas.DataFrame* method), 484  
 cummin() (*databricks.koalas.groupby.GroupBy*  
     method), 755  
 cummin() (*databricks.koalas.Series* method), 163  
 cumprod() (*databricks.koalas.DataFrame* method),  
     488  
 cumprod() (*databricks.koalas.groupby.GroupBy*  
     method), 756  
 cumprod() (*databricks.koalas.Series* method), 165  
 cumsum() (*databricks.koalas.DataFrame* method), 487  
 cumsum() (*databricks.koalas.groupby.GroupBy*  
     method), 757  
 cumsum() (*databricks.koalas.Series* method), 164

## D

data\_type() (*databricks.koalas.Index.spark* prop-

- erty), 654
- data\_type() (databricks.koalas.MultiIndex.spark property), 710
- data\_type() (databricks.koalas.Series.spark property), 260
- DataFrame (class in databricks.koalas), 341
- date() (databricks.koalas.Series.dt property), 263
- date\_range() (in module databricks.koalas), 93
- DatetimeIndex (class in databricks.koalas), 713
- day() (databricks.koalas.DatetimeIndex property), 718
- day() (databricks.koalas.Series.dt property), 263
- day\_name() (databricks.koalas.DatetimeIndex method), 729
- day\_name() (databricks.koalas.Series.dt method), 276
- day\_of\_week() (databricks.koalas.DatetimeIndex property), 720
- day\_of\_year() (databricks.koalas.DatetimeIndex property), 721
- dayofweek() (databricks.koalas.DatetimeIndex property), 719
- dayofweek() (databricks.koalas.Series.dt property), 264
- dayofyear() (databricks.koalas.DatetimeIndex property), 721
- dayofyear() (databricks.koalas.Series.dt property), 265
- days\_in\_month() (databricks.koalas.DatetimeIndex property), 724
- days\_in\_month() (databricks.koalas.Series.dt property), 271
- daysinmonth() (databricks.koalas.DatetimeIndex property), 724
- daysinmonth() (databricks.koalas.Series.dt property), 271
- decode() (databricks.koalas.Series.str method), 282
- delete() (databricks.koalas.Index method), 630
- delete() (databricks.koalas.MultiIndex method), 693
- density() (databricks.koalas.DataFrame.plot method), 602
- density() (databricks.koalas.Series.plot method), 320
- describe() (databricks.koalas.DataFrame method), 466
- describe() (databricks.koalas.groupby.DataFrameGroupBy method), 774
- describe() (databricks.koalas.Series method), 166
- diff() (databricks.koalas.DataFrame method), 490
- diff() (databricks.koalas.groupby.GroupBy method), 765
- diff() (databricks.koalas.Series method), 194
- difference() (databricks.koalas.Index method), 660
- difference() (databricks.koalas.MultiIndex method), 704
- div() (databricks.koalas.DataFrame method), 393
- div() (databricks.koalas.Series method), 125
- divmod() (databricks.koalas.Series method), 137
- dot() (databricks.koalas.DataFrame method), 444
- dot() (databricks.koalas.Series method), 143
- drop() (databricks.koalas.DataFrame method), 499
- drop() (databricks.koalas.Index method), 636
- drop() (databricks.koalas.MultiIndex method), 692
- drop() (databricks.koalas.Series method), 201
- drop\_duplicates() (databricks.koalas.DataFrame method), 501
- drop\_duplicates() (databricks.koalas.Index method), 636
- drop\_duplicates() (databricks.koalas.Series method), 204
- droplevel() (databricks.koalas.DataFrame method), 500
- droplevel() (databricks.koalas.Index method), 646
- droplevel() (databricks.koalas.MultiIndex method), 687
- droplevel() (databricks.koalas.Series method), 203
- dropna() (databricks.koalas.DataFrame method), 523
- dropna() (databricks.koalas.Index method), 648
- dropna() (databricks.koalas.MultiIndex method), 688
- dropna() (databricks.koalas.Series method), 234
- dtype() (databricks.koalas.Index property), 621
- dtype() (databricks.koalas.Series property), 102
- dtypes() (databricks.koalas.DataFrame property), 350
- dtypes() (databricks.koalas.Series property), 102
- duplicated() (databricks.koalas.DataFrame method), 503
- ## E
- empty() (databricks.koalas.DataFrame property), 349
- empty() (databricks.koalas.Index property), 624
- empty() (databricks.koalas.MultiIndex property), 684
- empty() (databricks.koalas.Series property), 104
- encode() (databricks.koalas.Series.str method), 282
- endswith() (databricks.koalas.Series.str method), 282
- eq() (databricks.koalas.DataFrame method), 444
- eq() (databricks.koalas.Series method), 141
- equals() (databricks.koalas.DataFrame method), 504
- equals() (databricks.koalas.Index method), 631
- equals() (databricks.koalas.MultiIndex method), 690
- equals() (databricks.koalas.Series method), 205
- eval() (databricks.koalas.DataFrame method), 491
- expanding() (databricks.koalas.DataFrame method), 456
- expanding() (databricks.koalas.Series method), 152
- explain() (databricks.koalas.DataFrame.spark method), 590
- explode() (databricks.koalas.DataFrame method), 544
- explode() (databricks.koalas.Series method), 242



`extract()` (*databricks.koalas.Series.str method*), 283  
`extractall()` (*databricks.koalas.Series.str method*), 283

## F

`factorize()` (*databricks.koalas.Index method*), 631  
`factorize()` (*databricks.koalas.Series method*), 245  
`ffill()` (*databricks.koalas.DataFrame method*), 529  
`ffill()` (*databricks.koalas.groupby.GroupBy method*), 770  
`fillna()` (*databricks.koalas.DataFrame method*), 525  
`fillna()` (*databricks.koalas.groupby.GroupBy method*), 768  
`fillna()` (*databricks.koalas.Index method*), 647  
`fillna()` (*databricks.koalas.MultiIndex method*), 688  
`fillna()` (*databricks.koalas.Series method*), 234  
`filter()` (*databricks.koalas.DataFrame method*), 504  
`filter()` (*databricks.koalas.groupby.GroupBy method*), 758  
`filter()` (*databricks.koalas.Series method*), 169  
`find()` (*databricks.koalas.Series.str method*), 283  
`findall()` (*databricks.koalas.Series.str method*), 284  
`first()` (*databricks.koalas.DataFrame method*), 505  
`first()` (*databricks.koalas.groupby.GroupBy method*), 759  
`first()` (*databricks.koalas.Series method*), 207  
`first_valid_index()` (*databricks.koalas.DataFrame method*), 565  
`first_valid_index()` (*databricks.koalas.Series method*), 255  
`Float64Index` (class in *databricks.koalas*), 666  
`floor()` (*databricks.koalas.DatetimeIndex method*), 727  
`floor()` (*databricks.koalas.Series.dt method*), 274  
`floordiv()` (*databricks.koalas.DataFrame method*), 435  
`floordiv()` (*databricks.koalas.Series method*), 136  
`frame()` (*databricks.koalas.DataFrame.spark method*), 583  
`from_arrays()` (*databricks.koalas.MultiIndex static method*), 678  
`from_frame()` (*databricks.koalas.MultiIndex static method*), 680  
`from_product()` (*databricks.koalas.MultiIndex static method*), 679  
`from_records()` (*databricks.koalas.DataFrame static method*), 569  
`from_tuples()` (*databricks.koalas.MultiIndex static method*), 678

## G

`ge()` (*databricks.koalas.DataFrame method*), 443  
`ge()` (*databricks.koalas.Series method*), 140

`get()` (*databricks.koalas.DataFrame method*), 379  
`get()` (*databricks.koalas.Series method*), 122  
`get()` (*databricks.koalas.Series.str method*), 286  
`get_dummies()` (*databricks.koalas.Series.str method*), 287  
`get_dummies()` (in module *databricks.koalas*), 78  
`get_group()` (*databricks.koalas.groupby.GroupBy method*), 741  
`get_option()` (in module *databricks.koalas*), 73  
`groupby()` (*databricks.koalas.DataFrame method*), 454  
`groupby()` (*databricks.koalas.Series method*), 151  
`gt()` (*databricks.koalas.DataFrame method*), 442  
`gt()` (*databricks.koalas.Series method*), 139

## H

`has_duplicates()` (*databricks.koalas.Index property*), 620  
`has_duplicates()` (*databricks.koalas.MultiIndex property*), 681  
`hasnans()` (*databricks.koalas.Index property*), 620  
`hasnans()` (*databricks.koalas.MultiIndex property*), 682  
`hasnans()` (*databricks.koalas.Series property*), 105  
`head()` (*databricks.koalas.DataFrame method*), 363  
`head()` (*databricks.koalas.groupby.GroupBy method*), 770  
`head()` (*databricks.koalas.Series method*), 208  
`hint()` (*databricks.koalas.DataFrame.spark method*), 587  
`hist()` (*databricks.koalas.DataFrame method*), 603  
`hist()` (*databricks.koalas.DataFrame.plot method*), 598  
`hist()` (*databricks.koalas.Series method*), 325  
`hist()` (*databricks.koalas.Series.plot method*), 321  
`hour()` (*databricks.koalas.DatetimeIndex property*), 718  
`hour()` (*databricks.koalas.Series.dt property*), 263

## I

`iat()` (*databricks.koalas.DataFrame property*), 363  
`iat()` (*databricks.koalas.Series property*), 110  
`identical()` (*databricks.koalas.Index method*), 633  
`identical()` (*databricks.koalas.MultiIndex method*), 690  
`idxmax()` (*databricks.koalas.DataFrame method*), 364  
`idxmax()` (*databricks.koalas.groupby.GroupBy method*), 766  
`idxmax()` (*databricks.koalas.Series method*), 208  
`idxmin()` (*databricks.koalas.DataFrame method*), 365  
`idxmin()` (*databricks.koalas.groupby.GroupBy method*), 767  
`idxmin()` (*databricks.koalas.Series method*), 210  
`iloc()` (*databricks.koalas.DataFrame property*), 370



- `iloc()` (*databricks.koalas.Series* property), 114
- `Index` (class in *databricks.koalas*), 611
- `index()` (*databricks.koalas.DataFrame* property), 349
- `index()` (*databricks.koalas.Series* property), 102
- `index()` (*databricks.koalas.Series.str* method), 287
- `indexer_at_time()` (*databricks.koalas.DatetimeIndex* method), 725
- `indexer_between_time()` (*databricks.koalas.DatetimeIndex* method), 724
- `inferred_type()` (*databricks.koalas.Index* property), 621
- `inferred_type()` (*databricks.koalas.MultiIndex* property), 682
- `info()` (*databricks.koalas.DataFrame* method), 570
- `insert()` (*databricks.koalas.DataFrame* method), 563
- `insert()` (*databricks.koalas.Index* method), 634
- `insert()` (*databricks.koalas.MultiIndex* method), 691
- `Int64Index` (class in *databricks.koalas*), 663
- `intersection()` (*databricks.koalas.Index* method), 659
- `intersection()` (*databricks.koalas.MultiIndex* method), 703
- `is_all_dates()` (*databricks.koalas.Index* property), 622
- `is_all_dates()` (*databricks.koalas.MultiIndex* property), 682
- `is_boolean()` (*databricks.koalas.Index* method), 634
- `is_categorical()` (*databricks.koalas.Index* method), 634
- `is_floating()` (*databricks.koalas.Index* method), 635
- `is_integer()` (*databricks.koalas.Index* method), 635
- `is_interval()` (*databricks.koalas.Index* method), 635
- `is_leap_year()` (*databricks.koalas.DatetimeIndex* property), 723
- `is_leap_year()` (*databricks.koalas.Series.dt* property), 270
- `is_monotonic()` (*databricks.koalas.Index* property), 615
- `is_monotonic()` (*databricks.koalas.Series* property), 195
- `is_monotonic_decreasing()` (*databricks.koalas.Index* property), 618
- `is_monotonic_decreasing()` (*databricks.koalas.Series* property), 198
- `is_monotonic_increasing()` (*databricks.koalas.Index* property), 616
- `is_monotonic_increasing()` (*databricks.koalas.Series* property), 196
- `is_month_end()` (*databricks.koalas.DatetimeIndex* property), 721
- `is_month_end()` (*databricks.koalas.Series.dt* property), 266
- `is_month_start()` (*databricks.koalas.DatetimeIndex* property), 721
- `is_month_start()` (*databricks.koalas.Series.dt* property), 266
- `is_numeric()` (*databricks.koalas.Index* method), 635
- `is_object()` (*databricks.koalas.Index* method), 636
- `is_quarter_end()` (*databricks.koalas.DatetimeIndex* property), 722
- `is_quarter_end()` (*databricks.koalas.Series.dt* property), 268
- `is_quarter_start()` (*databricks.koalas.DatetimeIndex* property), 722
- `is_quarter_start()` (*databricks.koalas.Series.dt* property), 267
- `is_unique()` (*databricks.koalas.Index* property), 619
- `is_unique()` (*databricks.koalas.Series* property), 180
- `is_year_end()` (*databricks.koalas.DatetimeIndex* property), 723
- `is_year_end()` (*databricks.koalas.Series.dt* property), 269
- `is_year_start()` (*databricks.koalas.DatetimeIndex* property), 723
- `is_year_start()` (*databricks.koalas.Series.dt* property), 269
- `isalnum()` (*databricks.koalas.Series.str* method), 287
- `isalpha()` (*databricks.koalas.Series.str* method), 288
- `isdecimal()` (*databricks.koalas.Series.str* method), 292
- `isdigit()` (*databricks.koalas.Series.str* method), 288
- `isin()` (*databricks.koalas.DataFrame* method), 518
- `isin()` (*databricks.koalas.Index* method), 662
- `isin()` (*databricks.koalas.Series* method), 211
- `islower()` (*databricks.koalas.Series.str* method), 290
- `isna()` (*databricks.koalas.DataFrame* method), 356
- `isna()` (*databricks.koalas.Index* method), 649
- `isna()` (*databricks.koalas.Series* method), 230
- `isna()` (in module *databricks.koalas*), 86
- `isnull()` (*databricks.koalas.DataFrame* method), 357
- `isnull()` (*databricks.koalas.Series* method), 230
- `isnull()` (in module *databricks.koalas*), 87
- `isnumeric()` (*databricks.koalas.Series.str* method), 291
- `isspace()` (*databricks.koalas.Series.str* method), 289
- `istitle()` (*databricks.koalas.Series.str* method), 290
- `isupper()` (*databricks.koalas.Series.str* method), 290
- `item()` (*databricks.koalas.Index* method), 651
- `item()` (*databricks.koalas.MultiIndex* method), 707
- `item()` (*databricks.koalas.Series* method), 121
- `items()` (*databricks.koalas.DataFrame* method), 373
- `items()` (*databricks.koalas.Series* method), 120

`iteritems()` (*databricks.koalas.DataFrame method*), 373  
`iteritems()` (*databricks.koalas.Series method*), 120  
`iterrows()` (*databricks.koalas.DataFrame method*), 373  
`itertuples()` (*databricks.koalas.DataFrame method*), 374

## J

`join()` (*databricks.koalas.DataFrame method*), 560  
`join()` (*databricks.koalas.Series.str method*), 293

## K

`kde()` (*databricks.koalas.DataFrame method*), 603  
`kde()` (*databricks.koalas.Series.plot method*), 324  
`keys()` (*databricks.koalas.DataFrame method*), 375  
`keys()` (*databricks.koalas.Series method*), 117  
`kurt()` (*databricks.koalas.DataFrame method*), 469  
`kurt()` (*databricks.koalas.Series method*), 170  
`kurtosis()` (*databricks.koalas.DataFrame method*), 469  
`kurtosis()` (*databricks.koalas.Series method*), 189

## L

`last()` (*databricks.koalas.DataFrame method*), 506  
`last()` (*databricks.koalas.groupby.GroupBy method*), 759  
`last()` (*databricks.koalas.Series method*), 212  
`last_valid_index()` (*databricks.koalas.DataFrame method*), 567  
`last_valid_index()` (*databricks.koalas.Series method*), 256  
`le()` (*databricks.koalas.DataFrame method*), 443  
`le()` (*databricks.koalas.Series method*), 140  
`len()` (*databricks.koalas.Series.str method*), 293  
`levshape()` (*databricks.koalas.MultiIndex property*), 685  
`line()` (*databricks.koalas.DataFrame.plot method*), 599  
`line()` (*databricks.koalas.Series.plot method*), 322  
`ljust()` (*databricks.koalas.Series.str method*), 294  
`load_model()` (*in module databricks.koalas.mlflow*), 778  
`loc()` (*databricks.koalas.DataFrame property*), 366  
`loc()` (*databricks.koalas.Series property*), 111  
`local_checkpoint()` (*databricks.koalas.DataFrame.spark method*), 594  
`lower()` (*databricks.koalas.Series.str method*), 294  
`lstrip()` (*databricks.koalas.Series.str method*), 295  
`lt()` (*databricks.koalas.DataFrame method*), 442  
`lt()` (*databricks.koalas.Series method*), 139

## M

`mad()` (*databricks.koalas.DataFrame method*), 470  
`mad()` (*databricks.koalas.Series method*), 171  
`map()` (*databricks.koalas.Series method*), 149  
`map_in_pandas()` (*databricks.koalas.DataFrame method*), 458  
`mask()` (*databricks.koalas.DataFrame method*), 382  
`mask()` (*databricks.koalas.Series method*), 224  
`match()` (*databricks.koalas.Series.str method*), 296  
`max()` (*databricks.koalas.DataFrame method*), 470  
`max()` (*databricks.koalas.groupby.GroupBy method*), 759  
`max()` (*databricks.koalas.Index method*), 637  
`max()` (*databricks.koalas.MultiIndex method*), 698  
`max()` (*databricks.koalas.Series method*), 171  
`max()` (*databricks.koalas.window.Expanding method*), 740  
`max()` (*databricks.koalas.window.Rolling method*), 734  
`mean()` (*databricks.koalas.DataFrame method*), 471  
`mean()` (*databricks.koalas.groupby.GroupBy method*), 759  
`mean()` (*databricks.koalas.Series method*), 172  
`mean()` (*databricks.koalas.window.Expanding method*), 740  
`mean()` (*databricks.koalas.window.Rolling method*), 735  
`median()` (*databricks.koalas.DataFrame method*), 473  
`median()` (*databricks.koalas.groupby.GroupBy method*), 760  
`median()` (*databricks.koalas.Series method*), 186  
`melt()` (*databricks.koalas.DataFrame method*), 543  
`melt()` (*in module databricks.koalas*), 74  
`merge()` (*databricks.koalas.DataFrame method*), 557  
`merge()` (*in module databricks.koalas*), 76  
`microsecond()` (*databricks.koalas.DatetimeIndex property*), 719  
`microsecond()` (*databricks.koalas.Series.dt property*), 264  
`min()` (*databricks.koalas.DataFrame method*), 472  
`min()` (*databricks.koalas.groupby.GroupBy method*), 761  
`min()` (*databricks.koalas.Index method*), 637  
`min()` (*databricks.koalas.MultiIndex method*), 698  
`min()` (*databricks.koalas.Series method*), 173  
`min()` (*databricks.koalas.window.Expanding method*), 739  
`min()` (*databricks.koalas.window.Rolling method*), 732  
`minute()` (*databricks.koalas.DatetimeIndex property*), 718  
`minute()` (*databricks.koalas.Series.dt property*), 263  
`mod()` (*databricks.koalas.DataFrame method*), 428  
`mod()` (*databricks.koalas.Series method*), 134  
`mode()` (*databricks.koalas.Series method*), 173

- month() (*databricks.koalas.DatetimeIndex* property), 718
- month() (*databricks.koalas.Series.dt* property), 263
- month\_name() (*databricks.koalas.DatetimeIndex* method), 728
- month\_name() (*databricks.koalas.Series.dt* method), 276
- mul() (*databricks.koalas.DataFrame* method), 407
- mul() (*databricks.koalas.Series* method), 126
- MultiIndex (class in *databricks.koalas*), 674
- ## N
- name() (*databricks.koalas.Index* property), 623
- name() (*databricks.koalas.Series* property), 103
- names() (*databricks.koalas.Index* property), 623
- names() (*databricks.koalas.MultiIndex* property), 683
- ndim() (*databricks.koalas.DataFrame* property), 351
- ndim() (*databricks.koalas.Index* property), 623
- ndim() (*databricks.koalas.MultiIndex* property), 683
- ndim() (*databricks.koalas.Series* property), 103
- ne() (*databricks.koalas.DataFrame* method), 443
- ne() (*databricks.koalas.Series* method), 141
- nlargest() (*databricks.koalas.DataFrame* method), 537
- nlargest() (*databricks.koalas.groupby.SeriesGroupBy* method), 776
- nlargest() (*databricks.koalas.Series* method), 175
- nlevels() (*databricks.koalas.Index* property), 624
- nlevels() (*databricks.koalas.MultiIndex* property), 685
- normalize() (*databricks.koalas.DatetimeIndex* method), 726
- normalize() (*databricks.koalas.Series.dt* method), 271
- normalize() (*databricks.koalas.Series.str* method), 297
- notna() (*databricks.koalas.DataFrame* method), 358
- notna() (*databricks.koalas.Index* method), 649
- notna() (*databricks.koalas.Series* method), 231
- notna() (in module *databricks.koalas*), 88
- notnull() (*databricks.koalas.DataFrame* method), 359
- notnull() (*databricks.koalas.Series* method), 232
- notnull() (in module *databricks.koalas*), 90
- nsmallest() (*databricks.koalas.DataFrame* method), 538
- nsmallest() (*databricks.koalas.groupby.SeriesGroupBy* method), 775
- nsmallest() (*databricks.koalas.Series* method), 176
- nullable() (*databricks.koalas.Series.spark* property), 260
- nunique() (*databricks.koalas.DataFrame* method), 479
- nunique() (*databricks.koalas.groupby.GroupBy* method), 763
- nunique() (*databricks.koalas.Index* method), 642
- nunique() (*databricks.koalas.Series* method), 179
- ## O
- option\_context() (in module *databricks.koalas*), 74
- ordered() (*databricks.koalas.CategoricalIndex* property), 674
- ordered() (*databricks.koalas.Series.cat* property), 314
- ## P
- pad() (*databricks.koalas.DataFrame* method), 359
- pad() (*databricks.koalas.Series* method), 232
- pad() (*databricks.koalas.Series.str* method), 297
- partition() (*databricks.koalas.Series.str* method), 298
- pct\_change() (*databricks.koalas.DataFrame* method), 474
- pct\_change() (*databricks.koalas.Series* method), 177
- persist() (*databricks.koalas.DataFrame.spark* method), 585
- pie() (*databricks.koalas.DataFrame.plot* method), 600
- pie() (*databricks.koalas.Series.plot* method), 323
- pipe() (*databricks.koalas.DataFrame* method), 450
- pipe() (*databricks.koalas.Series* method), 153
- pivot() (*databricks.koalas.DataFrame* method), 533
- pivot\_table() (*databricks.koalas.DataFrame* method), 531
- plot (in module *databricks.koalas.DataFrame*), 595
- plot (in module *databricks.koalas.Series*), 316
- pop() (*databricks.koalas.DataFrame* method), 376
- pop() (*databricks.koalas.Series* method), 118
- pow() (*databricks.koalas.DataFrame* method), 421
- pow() (*databricks.koalas.Series* method), 132
- print\_schema() (*databricks.koalas.DataFrame.spark* method), 582
- prod() (*databricks.koalas.DataFrame* method), 475
- prod() (*databricks.koalas.Series* method), 178
- product() (*databricks.koalas.DataFrame* method), 476
- product() (*databricks.koalas.Series* method), 142
- PythonModelWrapper (class in *databricks.koalas.mlflow*), 778
- ## Q
- quantile() (*databricks.koalas.DataFrame* method), 478
- quantile() (*databricks.koalas.Series* method), 180
- quarter() (*databricks.koalas.DatetimeIndex* property), 721
- quarter() (*databricks.koalas.Series.dt* property), 266

`query()` (*databricks.koalas.DataFrame method*), 384

## R

`radd()` (*databricks.koalas.DataFrame method*), 389

`radd()` (*databricks.koalas.Series method*), 127

`range()` (in module *databricks.koalas*), 41

`rank()` (*databricks.koalas.DataFrame method*), 554

`rank()` (*databricks.koalas.groupby.GroupBy method*), 761

`rank()` (*databricks.koalas.Series method*), 181

`rdiv()` (*databricks.koalas.DataFrame method*), 396

`rdiv()` (*databricks.koalas.Series method*), 127

`rdivmod()` (*databricks.koalas.Series method*), 138

`read_clipboard()` (in module *databricks.koalas*), 57

`read_csv()` (in module *databricks.koalas*), 54

`read_delta()` (in module *databricks.koalas*), 44

`read_excel()` (in module *databricks.koalas*), 59

`read_html()` (in module *databricks.koalas*), 67

`read_json()` (in module *databricks.koalas*), 64

`read_orc()` (in module *databricks.koalas*), 49

`read_parquet()` (in module *databricks.koalas*), 47

`read_spark_io()` (in module *databricks.koalas*), 51

`read_sql()` (in module *databricks.koalas*), 71

`read_sql_query()` (in module *databricks.koalas*), 71

`read_sql_table()` (in module *databricks.koalas*), 70

`read_table()` (in module *databricks.koalas*), 42

`register_dataframe_accessor()` (in module *databricks.koalas.extensions*), 780

`register_index_accessor()` (in module *databricks.koalas.extensions*), 783

`register_series_accessor()` (in module *databricks.koalas.extensions*), 781

`reindex()` (*databricks.koalas.DataFrame method*), 550

`reindex()` (*databricks.koalas.Series method*), 214

`reindex_like()` (*databricks.koalas.DataFrame method*), 553

`reindex_like()` (*databricks.koalas.Series method*), 216

`rename()` (*databricks.koalas.DataFrame method*), 507

`rename()` (*databricks.koalas.Index method*), 638

`rename()` (*databricks.koalas.MultiIndex method*), 694

`rename()` (*databricks.koalas.Series method*), 213

`rename_axis()` (*databricks.koalas.DataFrame method*), 508

`rename_axis()` (*databricks.koalas.Series method*), 213

`repartition()` (*databricks.koalas.DataFrame.spark method*), 592

`repeat()` (*databricks.koalas.Index method*), 639

`repeat()` (*databricks.koalas.MultiIndex method*), 695

`repeat()` (*databricks.koalas.Series method*), 242

`repeat()` (*databricks.koalas.Series.str method*), 298

`replace()` (*databricks.koalas.DataFrame method*), 526

`replace()` (*databricks.koalas.Series method*), 248

`replace()` (*databricks.koalas.Series.str method*), 298

`reset_index()` (*databricks.koalas.DataFrame method*), 510

`reset_index()` (*databricks.koalas.Series method*), 217

`reset_option()` (in module *databricks.koalas*), 73

`rfind()` (*databricks.koalas.Series.str method*), 300

`rfloordiv()` (*databricks.koalas.DataFrame method*), 438

`rfloordiv()` (*databricks.koalas.Series method*), 137

`rindex()` (*databricks.koalas.Series.str method*), 301

`rjust()` (*databricks.koalas.Series.str method*), 301

`rmod()` (*databricks.koalas.DataFrame method*), 431

`rmod()` (*databricks.koalas.Series method*), 135

`rmul()` (*databricks.koalas.DataFrame method*), 410

`rmul()` (*databricks.koalas.Series method*), 128

`rolling()` (*databricks.koalas.DataFrame method*), 455

`rolling()` (*databricks.koalas.Series method*), 152

`round()` (*databricks.koalas.DataFrame method*), 489

`round()` (*databricks.koalas.DatetimeIndex method*), 727

`round()` (*databricks.koalas.Series method*), 193

`round()` (*databricks.koalas.Series.dt method*), 273

`rpartition()` (*databricks.koalas.Series.str method*), 302

`rpow()` (*databricks.koalas.DataFrame method*), 424

`rpow()` (*databricks.koalas.Series method*), 133

`rsplit()` (*databricks.koalas.Series.str method*), 302

`rstrip()` (*databricks.koalas.Series.str method*), 304

`rsub()` (*databricks.koalas.DataFrame method*), 417

`rsub()` (*databricks.koalas.Series method*), 129

`rtruediv()` (*databricks.koalas.DataFrame method*), 403

`rtruediv()` (*databricks.koalas.Series method*), 130

## S

`sample()` (*databricks.koalas.DataFrame method*), 519

`sample()` (*databricks.koalas.Series method*), 218

`scatter()` (*databricks.koalas.DataFrame.plot method*), 601

`schema()` (*databricks.koalas.DataFrame.spark method*), 582

`second()` (*databricks.koalas.DatetimeIndex property*), 719

`second()` (*databricks.koalas.Series.dt property*), 264

`select_dtypes()` (*databricks.koalas.DataFrame method*), 352

`sem()` (*databricks.koalas.DataFrame method*), 479



- `sem()` (*databricks.koalas.Series method*), 183  
`Series` (class in *databricks.koalas*), 95  
`set_index()` (*databricks.koalas.DataFrame method*), 513  
`set_names()` (*databricks.koalas.Index method*), 645  
`set_option()` (in module *databricks.koalas*), 73  
`shape()` (*databricks.koalas.DataFrame property*), 351  
`shape()` (*databricks.koalas.Index property*), 622  
`shape()` (*databricks.koalas.MultiIndex property*), 683  
`shape()` (*databricks.koalas.Series property*), 103  
`shift()` (*databricks.koalas.DataFrame method*), 564  
`shift()` (*databricks.koalas.groupby.GroupBy method*), 772  
`shift()` (*databricks.koalas.Index method*), 657  
`shift()` (*databricks.koalas.Series method*), 254  
`size()` (*databricks.koalas.DataFrame property*), 352  
`size()` (*databricks.koalas.groupby.GroupBy method*), 764  
`size()` (*databricks.koalas.Index property*), 624  
`size()` (*databricks.koalas.MultiIndex property*), 685  
`size()` (*databricks.koalas.Series property*), 104  
`skew()` (*databricks.koalas.DataFrame method*), 481  
`skew()` (*databricks.koalas.Series method*), 184  
`slice()` (*databricks.koalas.Series.str method*), 305  
`slice_replace()` (*databricks.koalas.Series.str method*), 306  
`sort_index()` (*databricks.koalas.DataFrame method*), 534  
`sort_index()` (*databricks.koalas.Series method*), 238  
`sort_values()` (*databricks.koalas.DataFrame method*), 536  
`sort_values()` (*databricks.koalas.Index method*), 655  
`sort_values()` (*databricks.koalas.MultiIndex method*), 712  
`sort_values()` (*databricks.koalas.Series method*), 239  
`split()` (*databricks.koalas.Series.str method*), 307  
`sql()` (in module *databricks.koalas*), 82  
`squeeze()` (*databricks.koalas.DataFrame method*), 545  
`squeeze()` (*databricks.koalas.Series method*), 243  
`stack()` (*databricks.koalas.DataFrame method*), 539  
`startswith()` (*databricks.koalas.Series.str method*), 309  
`std()` (*databricks.koalas.DataFrame method*), 482  
`std()` (*databricks.koalas.groupby.GroupBy method*), 762  
`std()` (*databricks.koalas.Series method*), 184  
`strftime()` (*databricks.koalas.DatetimeIndex method*), 726  
`strftime()` (*databricks.koalas.Series.dt method*), 272  
`strip()` (*databricks.koalas.Series.str method*), 310  
`style()` (*databricks.koalas.DataFrame property*), 581  
`sub()` (*databricks.koalas.DataFrame method*), 414  
`sub()` (*databricks.koalas.Series method*), 131  
`sum()` (*databricks.koalas.DataFrame method*), 481  
`sum()` (*databricks.koalas.groupby.GroupBy method*), 763  
`sum()` (*databricks.koalas.Series method*), 185  
`sum()` (*databricks.koalas.window.Expanding method*), 738  
`sum()` (*databricks.koalas.window.Rolling method*), 731  
`swapaxes()` (*databricks.koalas.DataFrame method*), 514  
`swapaxes()` (*databricks.koalas.Series method*), 220  
`swapcase()` (*databricks.koalas.Series.str method*), 310  
`swaplevel()` (*databricks.koalas.DataFrame method*), 515  
`swaplevel()` (*databricks.koalas.MultiIndex method*), 686  
`swaplevel()` (*databricks.koalas.Series method*), 219  
`symmetric_difference()` (*databricks.koalas.Index method*), 660  
`symmetric_difference()` (*databricks.koalas.MultiIndex method*), 704
- ## T
- `T()` (*databricks.koalas.DataFrame property*), 547  
`T()` (*databricks.koalas.Index property*), 625  
`T()` (*databricks.koalas.MultiIndex property*), 684  
`T()` (*databricks.koalas.Series property*), 105  
`tail()` (*databricks.koalas.DataFrame method*), 377  
`tail()` (*databricks.koalas.groupby.GroupBy method*), 773  
`tail()` (*databricks.koalas.Series method*), 222  
`take()` (*databricks.koalas.DataFrame method*), 517  
`take()` (*databricks.koalas.Index method*), 640  
`take()` (*databricks.koalas.MultiIndex method*), 696  
`take()` (*databricks.koalas.Series method*), 221  
`title()` (*databricks.koalas.Series.str method*), 311  
`to_clipboard()` (*databricks.koalas.DataFrame method*), 58  
`to_clipboard()` (*databricks.koalas.Series method*), 329  
`to_csv()` (*databricks.koalas.DataFrame method*), 55  
`to_csv()` (*databricks.koalas.Series method*), 334  
`to_datetime()` (in module *databricks.koalas*), 91  
`to_delta()` (*databricks.koalas.DataFrame method*), 45  
`to_dict()` (*databricks.koalas.DataFrame method*), 575  
`to_dict()` (*databricks.koalas.Series method*), 328  
`to_excel()` (*databricks.koalas.DataFrame method*), 62  
`to_excel()` (*databricks.koalas.Series method*), 337  
`to_frame()` (*databricks.koalas.Index method*), 652

- `to_frame()` (*databricks.koalas.MultiIndex method*), 708  
`to_frame()` (*databricks.koalas.Series method*), 338  
`to_html()` (*databricks.koalas.DataFrame method*), 68  
`to_json()` (*databricks.koalas.DataFrame method*), 65  
`to_json()` (*databricks.koalas.Series method*), 333  
`to_koalas()` (*databricks.koalas.DataFrame method*), 573  
`to_latex()` (*databricks.koalas.DataFrame method*), 579  
`to_latex()` (*databricks.koalas.Series method*), 330  
`to_list()` (*databricks.koalas.Index method*), 651  
`to_list()` (*databricks.koalas.MultiIndex method*), 707  
`to_list()` (*databricks.koalas.Series method*), 327  
`to_markdown()` (*databricks.koalas.DataFrame method*), 577  
`to_markdown()` (*databricks.koalas.Series method*), 332  
`to_numeric()` (*in module databricks.koalas*), 85  
`to_numpy()` (*databricks.koalas.DataFrame method*), 572  
`to_numpy()` (*databricks.koalas.Index method*), 653  
`to_numpy()` (*databricks.koalas.MultiIndex method*), 709  
`to_numpy()` (*databricks.koalas.Series method*), 326  
`to_orc()` (*databricks.koalas.DataFrame method*), 50  
`to_pandas()` (*databricks.koalas.DataFrame method*), 571  
`to_pandas()` (*databricks.koalas.Series method*), 326  
`to_parquet()` (*databricks.koalas.DataFrame method*), 48  
`to_records()` (*databricks.koalas.DataFrame method*), 578  
`to_series()` (*databricks.koalas.Index method*), 652  
`to_series()` (*databricks.koalas.MultiIndex method*), 708  
`to_spark()` (*databricks.koalas.DataFrame method*), 574  
`to_spark_io()` (*databricks.koalas.DataFrame method*), 52  
`to_spark_io()` (*databricks.koalas.DataFrame.spark method*), 588  
`to_string()` (*databricks.koalas.DataFrame method*), 574  
`to_string()` (*databricks.koalas.Series method*), 327  
`to_table()` (*databricks.koalas.DataFrame method*), 43  
`to_table()` (*databricks.koalas.DataFrame.spark method*), 587  
`transform()` (*databricks.koalas.DataFrame method*), 456  
`transform()` (*databricks.koalas.groupby.GroupBy method*), 745  
`transform()` (*databricks.koalas.Index.spark method*), 654  
`transform()` (*databricks.koalas.MultiIndex.spark method*), 711  
`transform()` (*databricks.koalas.Series method*), 148  
`transform()` (*databricks.koalas.Series.spark method*), 260  
`transform_batch()` (*databricks.koalas.DataFrame.koalas method*), 608  
`transform_batch()` (*databricks.koalas.Series.koalas method*), 339  
`translate()` (*databricks.koalas.Series.str method*), 311  
`transpose()` (*databricks.koalas.DataFrame method*), 549  
`truediv()` (*databricks.koalas.DataFrame method*), 400  
`truediv()` (*databricks.koalas.Series method*), 132  
`truncate()` (*databricks.koalas.DataFrame method*), 520  
`truncate()` (*databricks.koalas.Series method*), 225
- ## U
- `union()` (*databricks.koalas.Index method*), 659  
`union()` (*databricks.koalas.MultiIndex method*), 703  
`unique()` (*databricks.koalas.groupby.SeriesGroupBy method*), 777  
`unique()` (*databricks.koalas.Index method*), 641  
`unique()` (*databricks.koalas.MultiIndex method*), 697  
`unique()` (*databricks.koalas.Series method*), 190  
`unstack()` (*databricks.koalas.DataFrame method*), 541  
`unstack()` (*databricks.koalas.Series method*), 241  
`update()` (*databricks.koalas.DataFrame method*), 562  
`update()` (*databricks.koalas.Series method*), 251  
`upper()` (*databricks.koalas.Series.str method*), 312
- ## V
- `value_counts()` (*databricks.koalas.groupby.SeriesGroupBy method*), 776  
`value_counts()` (*databricks.koalas.Index method*), 643  
`value_counts()` (*databricks.koalas.MultiIndex method*), 699  
`value_counts()` (*databricks.koalas.Series method*), 190  
`values()` (*databricks.koalas.DataFrame property*), 354  
`values()` (*databricks.koalas.Index property*), 625  
`values()` (*databricks.koalas.MultiIndex property*), 686  
`values()` (*databricks.koalas.Series property*), 106  
`var()` (*databricks.koalas.DataFrame method*), 483

[var\(\)](#) (*databricks.koalas.groupby.GroupBy method*),  
[763](#)  
[var\(\)](#) (*databricks.koalas.Series method*), [188](#)  
[view\(\)](#) (*databricks.koalas.Index method*), [653](#)  
[view\(\)](#) (*databricks.koalas.MultiIndex method*), [709](#)

## W

[week\(\)](#) (*databricks.koalas.DatetimeIndex property*),  
[719](#)  
[week\(\)](#) (*databricks.koalas.Series.dt property*), [264](#)  
[weekday\(\)](#) (*databricks.koalas.DatetimeIndex property*), [720](#)  
[weekday\(\)](#) (*databricks.koalas.Series.dt property*), [265](#)  
[weekofyear\(\)](#) (*databricks.koalas.DatetimeIndex property*), [719](#)  
[weekofyear\(\)](#) (*databricks.koalas.Series.dt property*),  
[264](#)  
[where\(\)](#) (*databricks.koalas.DataFrame method*), [380](#)  
[where\(\)](#) (*databricks.koalas.Series method*), [223](#)  
[wrap\(\)](#) (*databricks.koalas.Series.str method*), [312](#)

## X

[xs\(\)](#) (*databricks.koalas.DataFrame method*), [378](#)  
[xs\(\)](#) (*databricks.koalas.Series method*), [121](#)

## Y

[year\(\)](#) (*databricks.koalas.DatetimeIndex property*),  
[718](#)  
[year\(\)](#) (*databricks.koalas.Series.dt property*), [263](#)

## Z

[zfill\(\)](#) (*databricks.koalas.Series.str method*), [313](#)