Modin

Release 0.12.1+0.g34962ec.dirty

Modin contributors

Dec 19, 2021

INSTALLATION

Modin uses Ray or Dask to provide an effortless way to speed up your pandas notebooks, scripts, and libraries. Unlike other distributed DataFrame libraries, Modin provides seamless integration and compatibility with existing pandas code. Even using the DataFrame constructor is identical.

```
import modin.pandas as pd
import numpy as np

frame_data = np.random.randint(0, 100, size=(2**10, 2**8))
df = pd.DataFrame(frame_data)
```

To use Modin, you do not need to know how many cores your system has and you do not need to specify how to distribute the data. In fact, you can continue using your previous pandas notebooks while experiencing a considerable speedup from Modin, even on a single machine. Once you've changed your import statement, you're ready to use Modin just like you would pandas.

CHAPTER

ONE

INSTALLATION AND CHOOSING YOUR COMPUTE ENGINE

Modin can be installed from PyPI:

```
pip install modin
```

If you don't have Ray or Dask installed, you will need to install Modin with one of the targets:

```
pip install "modin[ray]" # Install Modin dependencies and Ray to run on Ray
pip install "modin[dask]" # Install Modin dependencies and Dask to run on Dask
pip install "modin[all]" # Install all of the above
```

Modin will automatically detect which engine you have installed and use that for scheduling computation!

If you want to choose a specific compute engine to run on, you can set the environment variable `MODIN_ENGINE` and Modin will do computation with that engine:

```
export MODIN_ENGINE=ray  # Modin will use Ray
export MODIN_ENGINE=dask  # Modin will use Dask
```

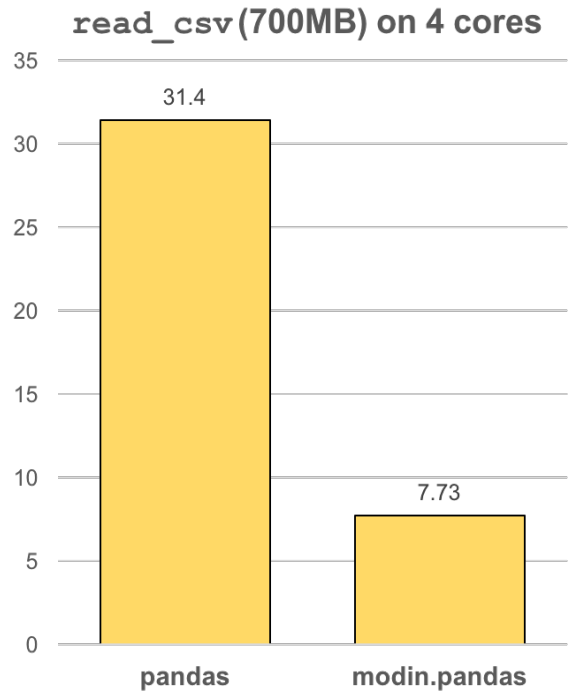
This can also be done within a notebook/interpreter before you import Modin:

```
import os

os.environ["MODIN_ENGINE"] = "ray"  # Modin will use Ray
os.environ["MODIN_ENGINE"] = "dask"  # Modin will use Dask

import modin.pandas as pd
```

CHAPTER

TWO

FASTER PANDAS, EVEN ON YOUR LAPTOP



The `modin.pandas` DataFrame is an extremely light-weight parallel DataFrame. Modin transparently distributes the data and computation so that all you need to do is continue using the pandas API as you were before installing Modin. Unlike other parallel DataFrame systems, Modin is an extremely light-weight, robust DataFrame. Because it is so light-weight, Modin provides speed-ups of up to 4x on a laptop with 4 physical cores.

In pandas, you are only able to use one core at a time when you are doing computation of any kind. With Modin, you are able to use all of the CPU cores on your machine. Even in `read_csv`, we see large gains by efficiently distributing the work across your entire machine.

```
import modin.pandas as pd

df = pd.read_csv("my_dataset.csv")
```

CHAPTER

THREE

MODIN IS A DATAFRAME FOR DATASETS FROM 1MB TO 1TB+

We have focused heavily on bridging the solutions between DataFrames for small data (e.g. pandas) and large data. Often data scientists require different tools for doing the same thing on different sizes of data. The DataFrame solutions that exist for 1MB do not scale to 1TB+, and the overheads of the solutions for 1TB+ are too costly for datasets in the 1KB range. With Modin, because of its light-weight, robust, and scalable nature, you get a fast DataFrame at 1MB and 1TB+.

Modin is currently under active development. Requests and contributions are welcome!

If you are interested in contributions please refer to 'developer documentation' section, where you can find 'Getting started' guide, system architecture and internal implementation details docs and lots of other useful information.

3.1 Installation

There are several ways to install Modin. Most users will want to install with `pip` or using `conda` tool, but some users may want to build from the master branch on the GitHub repo. The master branch has the most recent patches, but may be less stable than a release installed from `pip` or `conda`.

3.1.1 Installing with pip

Stable version

Modin can be installed with `pip` on Linux, Windows and MacOS. 2 engines are available for those platforms: *Ray* and *Dask* To install the most recent stable release run the following:

```
pip install -U modin # -U for upgrade in case you have an older version
```

If you don't have Ray or Dask installed, you will need to install Modin with one of the targets:

```
pip install modin[ray] # Install Modin dependencies and Ray to run on Ray
pip install modin[dask] # Install Modin dependencies and Dask to run on Dask
pip install modin[all] # Install all of the above
```

Modin will automatically detect which engine you have installed and use that for scheduling computation!

Release candidates

Before most major releases, we will upload a release candidate to test and check if there are any problems. If you would like to install a pre-release of Modin, run the following:

```
pip install --pre modin
```

These pre-releases are uploaded for dependencies and users to test their existing code to ensure that it still works. If you find something wrong, please raise an issue or email the bug reporter: bug_reports@modin.org.

Installing specific dependency sets

Modin has a number of specific dependency sets for running Modin on different execution engines and storage formats or for different functionalities of Modin. Here is a list of dependency sets for Modin:

```
pip install "modin[dask]" # If you want to use the Dask execution engine
```

Installing on Google Colab

Modin can be used with Google Colab via the `pip` command, by running the following code in a new cell:

```
!pip install modin[all]
```

Since Colab preloads several of Modin's dependencies by default, we need to restart the Colab environment once Modin is installed by either clicking on the "RESTART RUNTIME" button in the installation output or by run the following code:

```
# Post-install automatically kill and restart Colab environment
import os
os.kill(os.getpid(), 9)
```

Once you have restarted the Colab environment, you can use Modin in Colab in subsequent sessions.

Note that on the free version of Colab, there is a limit on the compute resource. To leverage the full power of Modin, you may have to upgrade to Colab Pro to get access to more compute resources.

3.1.2 Installing with conda

Using conda-forge channel

Modin releases can be installed using `conda` from conda-forge channel. Starting from 0.10.1 it is possible to install modin with chosen engine(s) alongside. Current options are:

Package name in conda-forge	Engine(s)	Supported OSs
modin	Dask	Linux, Windows, MacOS
modin-dask	Dask	Linux, Windows, MacOS
modin-ray	Ray	Linux, Windows
modin-omnisci	OmniSci	Linux
modin-all	Dask, Ray, OmniSci	Linux

So for installing Dask and Ray engines into conda environment following command should be used:

```
conda install -c conda-forge modin-ray modin-dask
```

All set of engines could be available in conda environment by specifying

```
conda install -c conda-forge modin-all
```

or explicitly

```
conda install -c conda-forge modin-ray modin-dask modin-omnisci
```

Using Intel® Distribution of Modin

With `conda` it is also possible to install Intel Distribution of Modin, a special version of Modin that is part of Intel® oneAPI AI Analytics Toolkit. This version of Modin is powered by *OmniSci* engine that contains a bunch of optimizations for Intel hardware. More details can be found on Intel Distribution of Modin page.

3.1.3 Installing from the GitHub master branch

If you'd like to try Modin using the most recent updates from the master branch, you can also use `pip`.

```
pip install git+https://github.com/modin-project/modin
```

This will install directly from the repo without you having to manually clone it! Please be aware that these changes have not made it into a release and may not be completely stable.

3.1.4 Windows

All Modin engines except *OmniSci* are available both on Windows and Linux as mentioned above. Default engine on Windows is *Ray*. It is also possible to use Windows Subsystem For Linux (WSL), but this is generally not recommended due to the limitations and poor performance of Ray on WSL, a roughly 2-3x cost.

3.1.5 Building Modin from Source

If you're planning on *contributing* to Modin, you will need to ensure that you are building Modin from the local repository that you are working off of. Occasionally, there are issues in overlapping Modin installs from pypi and from source. To avoid these issues, we recommend uninstalling Modin before you install from source:

```
pip uninstall modin
```

To build from source, you first must clone the repo. We recommend forking the repository first through the GitHub interface, then cloning as follows:

```
git clone https://github.com/<your-github-username>/modin.git
```

Once cloned, `cd` into the `modin` directory and use `pip` to install:

```
cd modin
pip install -e .
```

3.2 Using Modin

Modin is an early stage DataFrame library that wraps pandas and transparently distributes the data and computation, accelerating your pandas workflows with **one line of code change.** The user does not need to know how many cores their system has, nor do they need to specify how to distribute the data. In fact, users can **continue using their previous pandas notebooks** while experiencing a considerable speedup from Modin, even on a single machine. Only a modification of the import statement is needed, as we demonstrate below. Once you've changed your import statement, you're ready to use Modin just like you would pandas, since the API is identical to pandas.

3.2.1 Quickstart

```
# import pandas as pd
import modin.pandas as pd
```

That's it. You're ready to use Modin on your previous pandas notebooks.

We currently have most of the pandas API implemented and are working toward full functional parity with pandas (as well as even more tools and features).

3.2.2 Using Modin on a Single Node

In local (without a cluster) modin will create and manage a local (dask or ray) cluster for the execution

In order to use the most up-to-date version of Modin, please follow the instructions on the *installation page*.

Once you import the library, you should see something similar to the following output:

```
>>> import modin.pandas as pd

Waiting for redis server at 127.0.0.1:14618 to respond...
Waiting for redis server at 127.0.0.1:31410 to respond...
Starting local scheduler with the following resources: {'CPU': 4, 'GPU': 0}.

======================================================================
View the web UI at http://localhost:8889/notebooks/ray_ui36796.ipynb?
↪token=ac25867d62c4ae87941bc5a0ecd5f517dbf80bd8e9b04218
======================================================================
```

Once you have executed `import modin.pandas as pd`, you're ready to begin running your pandas pipeline as you were before.

3.2.3 APIs Supported

Please note, the API is not yet complete. For some methods, you may see the following:

```
NotImplementedError: To contribute to Modin, please visit github.com/modin-project/modin.
```

We have compiled a list of *currently supported methods*.

If you would like to request a particular method be implemented, feel free to open an issue. Before you open an issue please make sure that someone else has not already requested that functionality.

3.2.4 Connecting to a database for *read_sql*

To make pandas read from a SQL database, you have two options:

1) Pass a connection string, e.g. `postgresql://reader:NWDMCE5xdipIjRrp@hh-pgsql-public.ebi.ac.uk:5432/pfmegrnargs`
2) Pass an open database connection, e.g. for psycopg2, `psycopg2.connect("dbname=pfmegrnargs user=reader password=NWDMCE5xdipIjRrp host=hh-pgsql-public.ebi.ac.uk")`

The first option works with both Modin and pandas. If you try the second option in Modin, Modin will default to pandas because open database connections cannot be pickled. Pickling is required to send connection details to remote workers. To handle the unique requirements of distributed database access, Modin has a distributed database connection called `ModinDatabaseConnection`:

```
import modin.pandas as pd
from modin.db_conn import ModinDatabaseConnection
con = ModinDatabaseConnection(
    'psycopg2',
    host='hh-pgsql-public.ebi.ac.uk',
    dbname='pfmegrnargs',
    user='reader',
    password='NWDMCE5xdipIjRrp')
df = pd.read_sql("SELECT * FROM rnc_database",
            con,
        index_col=None,
        coerce_float=True,
        params=None,
        parse_dates=None,
        chunksize=None)
```

The `ModinDatabaseConnection` will save any arguments you supply it and forward them to the workers to make their own connections.

3.2.5 Using Modin on a Cluster (experimental)

Modin is able to utilize Ray's built-in autoscaled cluster. However, this usage is still under heavy development. To launch a Ray autoscaled cluster using Amazon Web Service (AWS), you can use the file *examples/cluster/aws_example.yaml* as the config file when launching an autoscaled Ray cluster. For the commands, refer to the autoscaler documentation.

We will provide a sample config file for private servers and other cloud service providers as we continue to develop and improve Modin's cluster support.

See more on the *Modin in the Cloud* documentation page.

3.2.6 Advanced usage (experimental)

In some cases, it may be useful to customize your Ray environment. Below, we have listed a few ways you can solve common problems in data management with Modin by customizing your Ray environment. It is possible to use any of Ray's initialization parameters, which are all found in Ray's documentation.

```
import ray
ray.init()
import modin.pandas as pd
```

Modin will automatically connect to the Ray instance that is already running. This way, you can customize your Ray environment for use in Modin!

Exceeding memory (Out of core pandas)

Modin experimentally supports out of core operations. See more on the *Out of Core* documentation page.

Reducing or limiting the resources Modin can use

By default, Modin will use all of the resources available on your machine. It is possible, however, to limit the amount of resources Modin uses to free resources for another task or user. Here is how you would limit the number of CPUs Modin used in your bash environment variables:

```
export MODIN_CPUS=4
```

You can also specify this in your python script with `os.environ`. **Make sure you update the CPUS before you import Modin!**:

```
import os
os.environ["MODIN_CPUS"] = "4"
import modin.pandas as pd
```

If you're using a specific engine and want more control over the environment Modin uses, you can start Ray or Dask in your environment and Modin will connect to it. **Make sure you start the environment before you import Modin!**

```
import ray
ray.init(num_cpus=4)
import modin.pandas as pd
```

Specifying `num_cpus` limits the number of processors that Modin uses. You may also specify more processors than you have available on your machine, however this will not improve the performance (and might end up hurting the performance of the system).

3.2.7 Examples

You can find an example on our recent blog post or on the Jupyter Notebook that we used to create the blog post.

3.3 Out of Core in Modin

If you are working with very large files or would like to exceed your memory, you may change the primary location of the DataFrame. If you would like to exceed memory, you can use your disk as an overflow for the memory.

3.3.1 Starting Modin with out of core enabled

Out of core is now enabled by default for both Ray and Dask engines.

3.3.2 Disabling Out of Core

Out of core is enabled by the compute engine selected. To disable it, start your preferred compute engine with the appropriate arguments. For example:

```
import modin.pandas as pd
import ray

ray.init(_plasma_directory="/tmp")  # setting to disable out of core in Ray
df = pd.read_csv("some.csv")
```

If you are using Dask, you have to modify local configuration files. Visit the Dask documentation on object spilling to see how.

3.3.3 Running an example with out of core

Before you run this, please make sure you follow the instructions listed above.

```
import modin.pandas as pd
import numpy as np
frame_data = np.random.randint(0, 100, size=(2**20, 2**8)) # 2GB each
df = pd.DataFrame(frame_data).add_prefix("col")
big_df = pd.concat([df for _ in range(20)]) # 20x2GB frames
print(big_df)
nan_big_df = big_df.isna() # The performance here represents a simple map
print(big_df.groupby("col1").count()) # group by on a large dataframe
```

This example creates a 40GB DataFrame from 20 identical 2GB DataFrames and performs various operations on them. Feel free to play around with this code and let us know what you think!

3.4 Examples

3.4.1 scikit-learn with LinearRegression

Here is a Jupyter Notebook example which uses Modin with scikit-learn and linear regression sklearn LinearRegression.

3.5 Overview

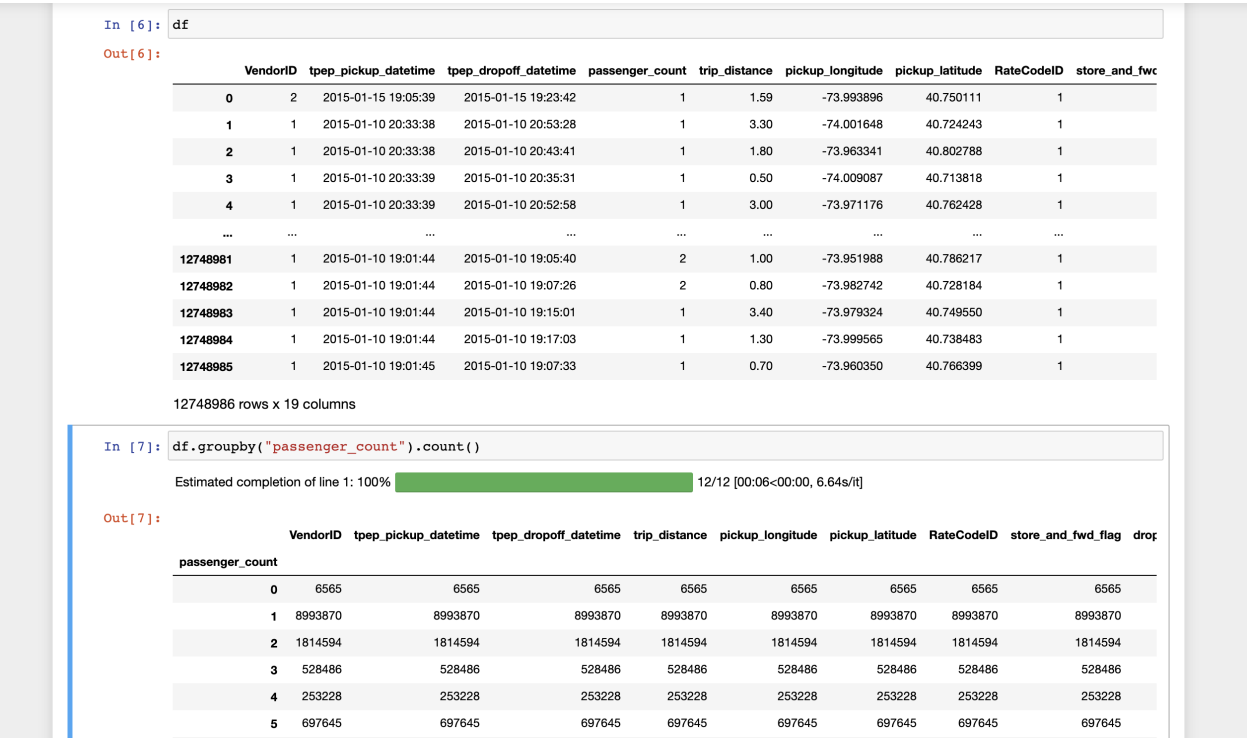
Modin aims to not only optimize Pandas, but also provide a comprehensive, integrated toolkit for data scientists. We are actively developing data science tools such as DataFrame - spreadsheet integration, DataFrame algebra, progress bars, SQL queries on DataFrames, and more. Join the Discourse for the latest updates!

3.5.1 Modin Spreadsheet API: Render Dataframes as Spreadsheets

The Spreadsheet API for Modin allows you to render the dataframe as a spreadsheet to easily explore your data and perform operations on a graphical user interface. The API also includes features for recording the changes made to the dataframe and exporting them as reproducible code. Built on top of Modin and SlickGrid, the spreadsheet interface is able to provide interactive response times even at a scale of billions of rows. See our Modin Spreadsheet API documentation for more details.

3.5.2 Progress Bar

Visual progress bar for Dataframe operations such as groupby and fillna, as well as for file reading operations such as read_csv. Built using the tqdm library and Ray execution engine. See Progress Bar documentation for more details.



In [6]:
```
df
```

Out[6]:

	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	pickup_longitude	pickup_latitude	RateCodeID	store_and_fwc
0	2	2015-01-15 19:05:39	2015-01-15 19:23:42	1	1.59	-73.993896	40.750111	1	
1	1	2015-01-10 20:33:38	2015-01-10 20:53:28	1	3.30	-74.001648	40.724243	1	
2	1	2015-01-10 20:33:38	2015-01-10 20:43:41	1	1.80	-73.963341	40.802788	1	
3	1	2015-01-10 20:33:39	2015-01-10 20:35:31	1	0.50	-74.009087	40.713818	1	
4	1	2015-01-10 20:33:39	2015-01-10 20:52:58	1	3.00	-73.971176	40.762428	1	
...	...	...	...	...	...	...	...	...	...
12748981	1	2015-01-10 19:01:44	2015-01-10 19:05:40	2	1.00	-73.951988	40.786217	1	
12748982	1	2015-01-10 19:01:44	2015-01-10 19:07:26	2	0.80	-73.982742	40.728184	1	
12748983	1	2015-01-10 19:01:44	2015-01-10 19:15:01	1	3.40	-73.979324	40.749550	1	
12748984	1	2015-01-10 19:01:44	2015-01-10 19:17:03	1	1.30	-73.999565	40.738483	1	
12748985	1	2015-01-10 19:01:45	2015-01-10 19:07:33	1	0.70	-73.960350	40.766399	1	

12748986 rows x 19 columns

In [7]:
```
df.groupby("passenger_count").count()
```

Estimated completion of line 1: 100% 12/12 [00:06<00:00, 6.64s/it]

Out[7]:

passenger_count	VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	trip_distance	pickup_longitude	pickup_latitude	RateCodeID	store_and_fwd_flag	drop
0	6565	6565	6565	6565	6565	6565	6565	6565	
1	8993870	8993870	8993870	8993870	8993870	8993870	8993870	8993870	
2	1814594	1814594	1814594	1814594	1814594	1814594	1814594	1814594	
3	528486	528486	528486	528486	528486	528486	528486	528486	
4	253228	253228	253228	253228	253228	253228	253228	253228	
5	697645	697645	697645	697645	697645	697645	697645	697645	

3.5.3 Dataframe Algebra

A minimal set of operators that can be composed to express any dataframe query for use in query planning and optimization. See our paper for more information, and full documentation is coming soon!

3.5.4 SQL on Modin Dataframes

```
# Importing in memory SQL engine as mdsql
# in the next couple of days you will import as follows
#  - import modin.experimental.sql as mdsql

# For now you can pip install the sql engine 'pip install pdsql' and include it as follo
from dfsql import sql_query
import modin.sql as mdsql
mdsql.query = sql_query

# You can then define the query that you want to perform
sql_str = "SELECT App,Category,Rating FROM gstore_apps WHERE Price = '0'"

# And simply apply that query to a dataframe
result_df = mdsql.query(sql_str, gstore_apps=gstore_apps_df)

# Or, in this case, where the query only requires one table,
# you can also ignore the FROM part in the query string:
query_str = "SELECT App, Category, Rating WHERE Price = '0' "

# mdsql.query can take query strings without FROM statement
# you can specify from as the function argument
result_df = mdsql.query(query_str, from=gstore_apps_df)
```

Read about Modin Dataframe support for SQL queries in this recent blog post. Check out the Modin SQL documentation as well!

3.5.5 Distributed XGBoost on Modin

Modin provides an implementation of distributed XGBoost machine learning algorithm on Modin DataFrames. See our *Distributed XGBoost on Modin documentation* for details about installation and usage, as well as *Modin XGBoost architecture documentation* for information about implementation and internal execution flow.

3.6 SQL on Modin Dataframes

MindsDB has teamed up with Modin to bring in-memory SQL to distributed Modin Dataframes. Now you can run SQL alongside the pandas API without copying or going through your disk. What this means is that you can now have a SQL solution that you can seamlessly scale horizontally and vertically, by leveraging the incredible power of Ray.

3.6.1 A Short Example Using the Google Play Store

```
import modin.pandas as pd
import modin.experimental.sql as sql

# read google play app store list from csv
gstore_apps_df = pd.read_csv("https://tinyurl.com/googleplaystorecsv")
```

	App	Category	Rating	Reviews	Size	Installs	Type	Price	Content Rating	Genres	Last Updated	Current Ver	Android Ver
0	Photo Editor & Candy Camera & Grid & ScrapBook	ART_AND_DESIGN	4.1	159	19M	10,000+	Free	0	Everyone	Art & Design	January 7, 2018	1.0.0	4.0.3 and up
1	Coloring book moana	ART_AND_DESIGN	3.9	967	14M	500,000+	Free	0	Everyone	Art & Design;Pretend Play	January 15, 2018	2.0.0	4.0.3 and up
2	U Launcher Lite – FREE Live Cool Themes, Hide ...	ART_AND_DESIGN	4.7	87510	8.7M	5,000,000+	Free	0	Everyone	Art & Design	August 1, 2018	1.2.4	4.0.3 and up
3	Sketch - Draw & Paint	ART_AND_DESIGN	4.5	215644	25M	50,000,000+	Free	0	Teen	Art & Design	June 8, 2018	Varies with device	4.2 and up

Imagine that you want to quickly select from 'gstore_apps_df' the columns App, Category, and Rating, where Price is '0'.

```
# You can then define the query that you want to perform
sql_str = "SELECT App, Category, Rating FROM gstore_apps WHERE Price = '0'"

# And simply apply that query to a dataframe
result_df = sql.query(sql_str, gstore_apps=gstore_apps_df)

# Or, in this case, where the query only requires one table,
# you can also ignore the FROM part in the query string:
query_str = "SELECT App, Category, Rating WHERE Price = '0' "

# sql.query can take query strings without FROM statement
# you can specify from as the function argument
result_df = sql.query(query_str, from=gstore_apps_df)
```

3.6.2 Writing Complex Queries

Let's explore a more complicated example.

```
gstore_reviews_df = pd.read_csv("https://tinyurl.com/gstorereviewscsv")
```

	App	Translated_Review	Sentiment	Sentiment_Polarity	Sentiment_Subjectivity
0	10 Best Foods for You	I like eat delicious food. That's I'm cooking ...	Positive	1.00	0.533333
1	10 Best Foods for You	This help eating healthy exercise regular basis	Positive	0.25	0.288462
2	10 Best Foods for You	NaN	NaN	NaN	NaN
3	10 Best Foods for You	Works great especially going grocery store	Positive	0.40	0.875000

Say we want to retrieve the top 10 app categories ranked by best average 'sentiment_polarity' where the average 'sentiment_subjectivity' is less than 0.5.

Since 'Category' is on the **gstore_apps_df** and sentiment_polarity is on **gstore_reviews_df**, we need to join the two tables, and operate averages on that join.

```
# Single query with join and group by
sql_str = """
SELECT
category,
avg(sentiment_polarity) as avg_sentiment_polarity,
avg(sentiment_subjectivity) as avg_sentiment_subjectivity
FROM (
SELECT
    category,
    CAST(sentiment as float) as sentiment,
    CAST(sentiment_polarity as float) as sentiment_polarity
FROM gstore_apps_df
    INNER JOIN gstore_reviews_df
    ON gstore_apps_df.app = gstore_reviews_df.app
) sub
GROUP BY category
HAVING avg_sentiment_subjectivity < 0.5
ORDER BY avg_sentiment_polarity DESC
LIMIT 10
"""

# Run query using apps and reviews dataframes,
# NOTE: that you simply pass the names of the tables in the query as arguments

result_df = sql.query( sql_str,
                       gstore_apps_df = gstore_apps_df,
                       gstore_reviews_df = gstore_reviews_df)
```

Or, you can bring the best of doing this in python and run the query in multiple parts (it's up to you).

```
# join the items and reviews

result_df = sql.query( """
SELECT
```

(continues on next page)

(continued from previous page)

```
    category,
    sentiment,
    sentiment_polarity
FROM gstore_apps_df INNER JOIN gstore_reviews_df
ON gstore_apps_df.app = gstore_reviews_df.app """,
gstore_apps_df = gstore_apps_df,
gstore_reviews_df = gstore_reviews_df )

# group by category and calculate averages

result_df = sql.query( """
SELECT
    category,
    avg(sentiment_polarity) as avg_sentiment_polarity,
    avg(sentiment_subjectivity) as avg_sentiment_subjectivity
GROUP BY category
HAVING CAST(avg_sentiment_subjectivity as float) < 0.5
ORDER BY avg_sentiment_polarity DESC
LIMIT 10""",
from = result_df)
```

If you have a cluster or even a computer with more than one CPU core, you can write SQL and Modin will run those queries in a distributed and optimized way.

3.6.3 Further Examples and Full Documentation

In the meantime, you can check out our Example Notebook that contains more examples and ideas, as well as this blog explaining Modin SQL usage.

3.7 Modin Spreadsheets API

3.7.1 Getting started

Install Modin-spreadsheet using pip:

```
pip install modin[spreadsheet]
```

The following code snippet creates a spreadsheet using the FiveThirtyEight dataset on labor force information by college majors (licensed under CC BY 4.0):

```
import modin.pandas as pd
import modin.spreadsheet as mss
df = pd.read_csv('https://raw.githubusercontent.com/fivethirtyeight/data/master/college-
↪majors/all-ages.csv')
spreadsheet = mss.from_dataframe(df)
spreadsheet
```

```
import modin.pandas as pd
import modin.spreadsheet as mss
df = pd.read_csv('https://raw.githubusercontent.com/fivethirtyeight/data/master/college-majors/all-ages.csv')
spreadsheet = mss.from_dataframe(df)
spreadsheet
```

Add Row | Remove Row | Clear History | Filter History | Reset Filters | Reset Sort

	Major_code	Major	Major_category	Total	Employed	Employed_fu
0	1100	GENERAL AGRICULT...	Agriculture & Natural ...	128148	90245	74078
1	1101	AGRICULTURE PROD...	Agriculture & Natural ...	95326	76865	64240
2	1102	AGRICULTURAL ECO...	Agriculture & Natural ...	33955	26321	22810
3	1103	ANIMAL SCIENCES	Agriculture & Natural ...	103549	81177	64937
4	1104	FOOD SCIENCE	Agriculture & Natural ...	24280	17281	12722
5	1105	PLANT SCIENCE AND...	Agriculture & Natural ...	79409	63043	51077
6	1106	SOIL SCIENCE	Agriculture & Natural ...	6586	4926	4042
7	1199	MISCELLANEOUS AG...	Agriculture & Natural ...	8549	6392	5074
8	1301	ENVIRONMENTAL SC...	Biology & Life Science	106106	87602	65238
9	1302	FORESTRY	Agriculture & Natural ...	69447	48228	39613
10	1303	NATURAL RESOURC...	Agriculture & Natural ...	83188	65937	50595
11	1401	ARCHITECTURE	Engineering	294692	216770	163020
12	1501	AREA ETHNIC AND C...	Humanities & Liberal ...	103740	75798	50530
13	1901	COMMUNICATIONS	Communications & Jo...	987676	790696	595739
14	1902	JOURNALISM	Communications & Jo...	418104	314438	235407

```
# ---- spreadsheet transformation history ----
unfiltered_df = df.copy()
```

3.7.2 Basic Manipulations through User Interface

The Spreadsheet API allows users to manipulate the DataFrame with simple graphical controls for sorting, filtering, and editing.

Here are the instructions for each operation:

- **Sort**: Click on the column header of the column to sort on.
- **Filter**: Click on the filter button on the column header and apply the desired filter to the column. The filter dropdown changes depending on the type of the column. Multiple filters are automatically combined.
- **Edit Cell**: Double click on a cell and enter the new value.
- **Add Rows**: Click on the "Add Row" button in the toolbar to duplicate the last row in the DataFrame. The duplicated values provide a convenient default and can be edited as necessary.
- **Remove Rows**: Select row(s) and click the "Remove Row" button. Select a single row by clicking on it. Multiple rows can be selected with Cmd+Click (Windows: Ctrl+Click) on the desired rows or with Shift+Click to specify a range of rows.

Some of these operations can also be done through the spreadsheet's programmatic interface. Sorts and filters can be reset using the toolbar buttons. Edits and adding/removing rows can only be undone manually.

3.7.3 Virtual Rendering

The spreadsheet will only render data based on the user's viewport. This allows for quick rendering even on very large DataFrames because only a handful of rows are loaded at any given time. As a result, scrolling and viewing your data is smooth and responsive.

3.7.4 Transformation History and Exporting Code

All operations on the spreadsheet are recorded and are easily exported as code for sharing or reproducibility. This history is automatically displayed in the history cell, which is generated below the spreadsheet whenever the spreadsheet widget is displayed. The history cell is displayed on default, but this can be turned off. Modin Spreadsheet API provides a few methods for interacting with the history:

- *SpreadsheetWidget.get_history()*: Retrieves the transformation history in the form of reproducible code.
- *SpreadsheetWidget.filter_relevant_history(persist=True)*: Returns the transformation history that contains only code relevant to the final state of the spreadsheet. The *persist* parameter determines whether the internal state and the displayed history is also filtered.
- *SpreadsheetWidget.reset_history()*: Clears the history of transformation.

3.7.5 Customizable Interface

The spreadsheet widget provides a number of options that allows the user to change the appearance and the interactivity of the spreadsheet. Options include:

- Row height/Column width
- Preventing edits, sorts, or filters on the whole spreadsheet or on a per-column basis
- Hiding the toolbar and history cell
- Float precision
- Highlighting of cells and rows
- Viewport size

3.7.6 Converting Spreadsheets To and From Dataframes

`modin.experimental.spreadsheet.general.`**`from_dataframe`**(*dataframe*, *show_toolbar=None*, *show_history=None*, *precision=None*, *grid_options=None*, *column_options=None*, *column_definitions=None*, *row_edit_callback=None*)

Renders a DataFrame or Series as an interactive spreadsheet, represented by an instance of the `SpreadsheetWidget` class. The `SpreadsheetWidget` instance is constructed using the options passed in to this function. The `dataframe` argument to this function is used as the `df` kwarg in call to the SpreadsheetWidget constructor, and the rest of the parameters are passed through as is.

If the `dataframe` argument is a Series, it will be converted to a DataFrame before being passed in to the SpreadsheetWidget constructor as the `df` kwarg.

Return type SpreadsheetWidget

Parameters

- **dataframe** (*DataFrame*) – The DataFrame that will be displayed by this instance of SpreadsheetWidget.
- **grid_options** (*dict*) – Options to use when creating the SlickGrid control (i.e. the interactive grid). See the Notes section below for more information on the available options, as well as the default options that this widget uses.
- **precision** (*integer*) – The number of digits of precision to display for floating-point values. If unset, we use the value of *pandas.get_option('display.precision')*.
- **show_toolbar** (*bool*) – Whether to show a toolbar with options for adding/removing rows. Adding/removing rows is an experimental feature which only works with DataFrames that have an integer index.
- **show_history** (*bool*) – Whether to show the cell containing the spreadsheet transformation history.
- **column_options** (*dict*) – Column options that are to be applied to every column. See the Notes section below for more information on the available options, as well as the default options that this widget uses.
- **column_definitions** (*dict*) – Column options that are to be applied to individual columns. The keys of the dict should be the column names, and each value should be the column options for a particular column, represented as a dict. The available options for each column are the same options that are available to be set for all columns via the `column_options` parameter. See the Notes section below for more information on those options.
- **row_edit_callback** (*callable*) – A callable that is called to determine whether a particular row should be editable or not. Its signature should be `callable(row)`, where `row` is a dictionary which contains a particular row's values, keyed by column name. The callback should return True if the provided row should be editable, and False otherwise.

Notes

The following dictionary is used for `grid_options` if none are provided explicitly:

```
{
    # SlickGrid options
    'fullWidthRows': True,
    'syncColumnCellResize': True,
    'forceFitColumns': False,
    'defaultColumnWidth': 150,
    'rowHeight': 28,
    'enableColumnReorder': False,
    'enableTextSelectionOnCells': True,
    'editable': True,
    'autoEdit': False,
    'explicitInitialization': True,

    # Modin-spreadsheet options
    'maxVisibleRows': 15,
    'minVisibleRows': 8,
    'sortable': True,
    'filterable': True,
    'highlightSelectedCell': False,
```

(continues on next page)

(continued from previous page)

```
    'highlightSelectedRow': True
}
```

The first group of options are SlickGrid "grid options" which are described in the SlickGrid documentation.

The second group of option are options that were added specifically for modin-spreadsheet and therefore are not documented in the SlickGrid documentation. The following bullet points describe these options.

- **maxVisibleRows** The maximum number of rows that modin-spreadsheet will show.
- **minVisibleRows** The minimum number of rows that modin-spreadsheet will show
- **sortable** Whether the modin-spreadsheet instance will allow the user to sort columns by clicking the column headers. When this is set to `False`, nothing will happen when users click the column headers.
- **filterable** Whether the modin-spreadsheet instance will allow the user to filter the grid. When this is set to `False` the filter icons won't be shown for any columns.
- **highlightSelectedCell** If you set this to True, the selected cell will be given a light blue border.
- **highlightSelectedRow** If you set this to False, the light blue background that's shown by default for selected rows will be hidden.

The following dictionary is used for `column_options` if none are provided explicitly:

```
{
    # SlickGrid column options
    'defaultSortAsc': True,
    'maxWidth': None,
    'minWidth': 30,
    'resizable': True,
    'sortable': True,
    'toolTip': "",
    'width': None

    # Modin-spreadsheet column options
    'editable': True,
}
```

The first group of options are SlickGrid "column options" which are described in the SlickGrid documentation.

The `editable` option was added specifically for modin-spreadsheet and therefore is not documented in the SlickGrid documentation. This option specifies whether a column should be editable or not.

See also:

`set_defaults` Permanently set global defaults for the parameters of `show_grid`, with the exception of the `dataframe` and `column_definitions` parameters, since those depend on the particular set of data being shown by an instance, and therefore aren't parameters we would want to set for all SpreadsheetWidget instances.

`set_grid_option` Permanently set global defaults for individual grid options. Does so by changing the defaults that the `show_grid` method uses for the `grid_options` parameter.

`SpreadsheetWidget` The widget class that is instantiated and returned by this method.

`modin.experimental.spreadsheet.general.`**`to_dataframe`**(*spreadsheet*)

Get a copy of the DataFrame that reflects the current state of the `spreadsheet` SpreadsheetWidget instance UI. This includes any sorting or filtering changes, as well as edits that have been made by double clicking cells.

Return type DataFrame

Parameters **spreadsheet** (*SpreadsheetWidget*) – The SpreadsheetWidget instance that DataFrame that will be displayed by this instance of SpreadsheetWidget.

3.7.7 Further API Documentation

class modin_spreadsheet.grid.**SpreadsheetWidget**(***kwargs*)

The widget class which is instantiated by the `show_grid` method. This class can be constructed directly but that's not recommended because then default options have to be specified explicitly (since default options are normally provided by the `show_grid` method).

The constructor for this class takes all the same parameters as `show_grid`, with one exception, which is that the required `data_frame` parameter is replaced by an optional keyword argument called `df`.

See also:

show_grid The method that should be used to construct SpreadsheetWidget instances, because it provides reasonable defaults for all of the modin-spreadsheet options.

df

Get/set the DataFrame that's being displayed by the current instance. This DataFrame will NOT reflect any sorting/filtering/editing changes that are made via the UI. To get a copy of the DataFrame that does reflect sorting/filtering/editing changes, use the `get_changed_df()` method.

Type DataFrame

grid_options

Get/set the grid options being used by the current instance.

Type dict

precision

Get/set the precision options being used by the current instance.

Type integer

show_toolbar

Get/set the show_toolbar option being used by the current instance.

Type bool

show_history

Get/set the show_history option being used by the current instance.

Type bool

column_options

Get/set the column options being used by the current instance.

Type bool

column_definitions

Get/set the column definitions (column-specific options) being used by the current instance.

Type bool

add_row(*row=None*)

Append a row at the end of the DataFrame. Values for the new row can be provided via the `row` argument, which is optional for DataFrames that have an integer index, and required otherwise. If the `row` argument is not provided, the last row will be duplicated and the index of the new row will be the index of the last row plus one.

Parameters **row** (*list (default: None)*) – A list of 2-tuples of (column name, column value) that specifies the values for the new row.

See also:

SpreadsheetWidget.remove_rows The method for removing a row (or rows).

change_grid_option(*option_name*, *option_value*)

Change a SlickGrid grid option without rebuilding the entire grid widget. Not all options are supported at this point so this method should be considered experimental.

Parameters

- **option_name** (*str*) – The name of the grid option to be changed.
- **option_value** (*str*) – The new value for the grid option.

change_selection(*rows=[]*)

Select a row (or rows) in the UI. The indices of the rows to select are provided via the optional `rows` argument.

Parameters **rows** (*list (default: [])*) – A list of indices of the rows to select. For a multi-indexed DataFrame, each index in the list should be a tuple, with each value in each tuple corresponding to a level of the MultiIndex. The default value of `[]` results in the no rows being selected (i.e. it clears the selection).

edit_cell(*index*, *column*, *value*)

Edit a cell of the grid, given the index and column of the cell to edit, as well as the new value of the cell. Results in a `cell_edited` event being fired.

Parameters

- **index** (*object*) – The index of the row containing the cell that is to be edited.
- **column** (*str*) – The name of the column containing the cell that is to be edited.
- **value** (*object*) – The new value for the cell.

get_changed_df()

Get a copy of the DataFrame that was used to create the current instance of SpreadsheetWidget which reflects the current state of the UI. This includes any sorting or filtering changes, as well as edits that have been made by double clicking cells.

Return type DataFrame

get_selected_df()

Get a DataFrame which reflects the current state of the UI and only includes the currently selected row(s). Internally it calls `get_changed_df()` and then filters down to the selected rows using `iloc`.

Return type DataFrame

get_selected_rows()

Get the currently selected rows.

Return type List of integers

off(*names*, *handler*)

Remove a modin-spreadsheet event handler that was registered with the current instance's `on` method.

Parameters

- **names** (*list, str, All (default: All)*) – The names of the events for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all events.

- **handler** (*callable*) – A callable that was previously registered with the current instance's on method.

See also:

SpreadsheetWidget.on The method for hooking up instance-level handlers that this off method can remove.

on(*names*, *handler*)

Setup a handler to be called when a user interacts with the current instance.

Parameters

- **names** (*list, str, All*) – If names is All, the handler will apply to all events. If a list of str, handler will apply to all events named in the list. If a str, the handler will apply just the event with that name.
- **handler** (*callable*) – A callable that is called when the event occurs. Its signature should be handler(event, spreadsheet_widget), where event is a dictionary and spreadsheet_widget is the SpreadsheetWidget instance that fired the event. The event dictionary at least holds a name key which specifies the name of the event that occurred.

Notes

Here's the list of events that you can listen to on SpreadsheetWidget instances via the on method:

```
[
    'cell_edited',
    'selection_changed',
    'viewport_changed',
    'row_added',
    'row_removed',
    'filter_dropdown_shown',
    'filter_changed',
    'sort_changed',
    'text_filter_viewport_changed',
    'json_updated'
]
```

The following bullet points describe the events listed above in more detail. Each event bullet point is followed by sub-bullets which describe the keys that will be included in the event dictionary for each event.

- **cell_edited** The user changed the value of a cell in the grid.
  - **index** The index of the row that contains the edited cell.
  - **column** The name of the column that contains the edited cell.
  - **old** The previous value of the cell.
  - **new** The new value of the cell.
- **filter_changed** The user changed the filter setting for a column.
  - **column** The name of the column for which the filter setting was changed.
- **filter_dropdown_shown** The user showed the filter control for a column by clicking the filter icon in the column's header.

  - **column** The name of the column for which the filter control was shown.

- **json_updated** A user action causes SpreadsheetWidget to send rows of data (in json format) down to the browser. This happens as a side effect of certain actions such as scrolling, sorting, and filtering.
  - **triggered_by** The name of the event that resulted in rows of data being sent down to the browser. Possible values are `change_viewport`, `change_filter`, `change_sort`, `add_row`, `remove_row`, and `edit_cell`.
  - **range** A tuple specifying the range of rows that have been sent down to the browser.
- **row_added** The user added a new row using the "Add Row" button in the grid toolbar.
  - **index** The index of the newly added row.
  - **source** The source of this event. Possible values are `api` (an api method call) and `gui` (the grid interface).
- **row_removed** The user added removed one or more rows using the "Remove Row" button in the grid toolbar.
  - **indices** The indices of the removed rows, specified as an array of integers.
  - **source** The source of this event. Possible values are `api` (an api method call) and `gui` (the grid interface).
- **selection_changed** The user changed which rows were highlighted in the grid.
  - **old** An array specifying the indices of the previously selected rows.
  - **new** The indices of the rows that are now selected, again specified as an array.
  - **source** The source of this event. Possible values are `api` (an api method call) and `gui` (the grid interface).
- **sort_changed** The user changed the sort setting for the grid.
  - **old** The previous sort setting for the grid, specified as a dict with the following keys:
    * **column** The name of the column that the grid was sorted by
    * **ascending** Boolean indicating ascending/descending order
  - **new** The new sort setting for the grid, specified as a dict with the following keys:
    * **column** The name of the column that the grid is currently sorted by
    * **ascending** Boolean indicating ascending/descending order
- **text_filter_viewport_changed** The user scrolled the new rows into view in the filter dropdown for a text field.
  - **column** The name of the column whose filter dropdown is visible
  - **old** A tuple specifying the previous range of visible rows in the filter dropdown.
  - **new** A tuple specifying the range of rows that are now visible in the filter dropdown.
- **viewport_changed** The user scrolled the new rows into view in the grid.
  - **old** A tuple specifying the previous range of visible rows.
  - **new** A tuple specifying the range of rows that are now visible.

The `event` dictionary for every type of event will contain a `name` key specifying the name of the event that occurred. That key is excluded from the lists of keys above to avoid redundacy.

See also:

on Same as the instance-level on method except it listens for events on all instances rather than on an individual SpreadsheetWidget instance.

SpreadsheetWidget.off Unhook a handler that was hooked up using the instance-level on method.

remove_row(*rows=None*)
Alias for remove_rows, which is provided for convenience because this was the previous name of that method.

remove_rows(*rows=None*)
Remove a row (or rows) from the DataFrame. The indices of the rows to remove can be provided via the optional rows argument. If the rows argument is not provided, the row (or rows) that are currently selected in the UI will be removed.

Parameters **rows** (*list (default: None)*) – A list of indices of the rows to remove from the DataFrame. For a multi-indexed DataFrame, each index in the list should be a tuple, with each value in each tuple corresponding to a level of the MultiIndex.

See also:

SpreadsheetWidget.add_row The method for adding a row.

SpreadsheetWidget.remove_row Alias for this method.

toggle_editable()
Change whether the grid is editable or not, without rebuilding the entire grid widget.

3.8 Progress Bar

The progress bar allows users to see the estimated progress and completion time of each line they run, in environments such as a shell or Jupyter notebook.

3.8.1 Quickstart

The progress bar uses the *tqdm* library to visualize displays:

```
pip install tqdm
```

Import the progress bar into your notebook by running the following:

```
import modin.pandas as pd
from tqdm import tqdm
from modin.config import ProgressBar
ProgressBar.enable()
```

3.9 Distributed XGBoost on Modin

Modin provides an implementation of distributed XGBoost machine learning algorithm on Modin DataFrames. Please note that this feature is experimental and behavior or interfaces could be changed.

3.9.1 Install XGBoost on Modin

Modin comes with all the dependencies except `xgboost` package by default. Currently, distributed XGBoost on Modin is only supported on the Ray execution engine, therefore, see the *installation page* for more information on installing Modin with the Ray engine. To install `xgboost` package you can use `pip`:

```
pip install xgboost
```

3.9.2 XGBoost Train and Predict

Distributed XGBoost functionality is placed in `modin.experimental.xgboost` module. `modin.experimental.xgboost` provides a drop-in replacement API for `train` and `Booster.predict` xgboost functions.

Module holds public interfaces for Modin XGBoost.

`modin.experimental.xgboost.train`(*params: Dict*, *dtrain:* modin.experimental.xgboost.xgboost.DMatrix, **args*, *evals=()*, *num_actors: Optional[int] = None*, *evals_result: Optional[Dict] = None*, ***kwargs*)

Run distributed training of XGBoost model.

During work it evenly distributes *dtrain* between workers according to IP addresses partitions (in case of not even distribution of *dtrain* over nodes, some partitions will be re-distributed between nodes), runs xgb.train on each worker for subset of *dtrain* and reduces training results of each worker using Rabit Context.

Parameters

- **params** (*dict*) – Booster params.
- **dtrain** (`modin.experimental.xgboost.DMatrix`) – Data to be trained against.
- ***args** (*iterable*) – Other parameters for *xgboost.train*.
- **evals** (*list of pairs (*`modin.experimental.xgboost.DMatrix`*, str), default: empty*) – List of validation sets for which metrics will evaluated during training. Validation metrics will help us track the performance of the model.
- **num_actors** (*int, optional*) – Number of actors for training. If unspecified, this value will be computed automatically.
- **evals_result** (*dict, optional*) – Dict to store evaluation results in.
- ****kwargs** (*dict*) – Other parameters are the same as *xgboost.train*.

Returns A trained booster.

Return type *modin.experimental.xgboost.Booster*

`class modin.experimental.xgboost.Booster`(*params=None*, *cache=()*, *model_file=None*)

A Modin Booster of XGBoost.

Booster is the model of XGBoost, that contains low level routines for training, prediction and evaluation.

Parameters

- **params** (*dict, optional*) – Parameters for boosters.

- **cache** (*list, default: empty*) – List of cache items.
- **model_file** (*string/os.PathLike/xgb.Booster/bytearray, optional*) – Path to the model file if it's string or PathLike or xgb.Booster.

predict(*data:* modin.experimental.xgboost.xgboost.DMatrix, ***kwargs*)

Run distributed prediction with a trained booster.

During execution it runs `xgb.predict` on each worker for subset of *data* and creates Modin DataFrame with prediction results.

Parameters

- **data** (modin.experimental.xgboost.DMatrix) – Input data used for prediction.
- ****kwargs** (*dict*) – Other parameters are the same as for `xgboost.Booster.predict`.

Returns Modin DataFrame with prediction results.

Return type modin.pandas.DataFrame

3.9.3 ModinDMatrix

Data is passed to `modin.experimental.xgboost` functions via a Modin `DMatrix` object.

Module holds public interfaces for Modin XGBoost.

class modin.experimental.xgboost.**DMatrix**(*data*, *label=None*, *missing=None*, *silent=False*, *feature_names=None*, *feature_types=None*, *feature_weights=None*, *enable_categorical=None*)

DMatrix holds references to partitions of Modin DataFrame.

On init stage unwrapping partitions of Modin DataFrame is started.

Parameters

- **data** (*modin.pandas.DataFrame*) – Data source of DMatrix.
- **label** (*modin.pandas.DataFrame or modin.pandas.Series, optional*) – Labels used for training.
- **missing** (*float, optional*) – Value in the input data which needs to be present as a missing value. If `None`, defaults to `np.nan`.
- **silent** (*boolean, optional*) – Whether to print messages during construction or not.
- **feature_names** (*list, optional*) – Set names for features.
- **feature_types** (*list, optional*) – Set types for features.
- **feature_weights** (*array_like, optional*) – Set feature weights for column sampling.
- **enable_categorical** (*boolean, optional*) – Experimental support of specializing for categorical features.

Notes

Currently DMatrix doesn't support *weight*, *base_margin*, *nthread*, *group*, *qid*, *label_lower_bound*, *label_upper_bound* parameters.

property feature_names
Get column labels.

Returns

Return type Column labels.

property feature_types
Get column types.

Returns

Return type Column types.

get_dmatrix_params()
Get dict of DMatrix parameters excluding *self.data*/*self.label*.

Returns

Return type dict

get_float_info(*name*)
Get float property from the DMatrix.

Parameters **name** (*str*) – The field name of the information.

Returns

Return type A NumPy array of float information of the data.

num_col()
Get number of columns.

Returns

Return type int

num_row()
Get number of rows.

Returns

Return type int

set_info(*, *label=None*, *feature_names=None*, *feature_types=None*, *feature_weights=None*) → None
Set meta info for DMatrix.

Parameters

- **label** (*modin.pandas.DataFrame or modin.pandas.Series, optional*) – Labels used for training.
- **feature_names** (*list, optional*) – Set names for features.
- **feature_types** (*list, optional*) – Set types for features.
- **feature_weights** (*array_like, optional*) – Set feature weights for column sampling.

Currently, the Modin `DMatrix` supports `modin.pandas.DataFrame` only as an input.

3.9.4 A Single Node / Cluster setup

The XGBoost part of Modin uses a Ray resources by similar way as all Modin functions.

To start the Ray runtime on a single node:

```
import ray
ray.init()
```

If you already had the Ray cluster you can connect to it by next way:

```
import ray
ray.init(address='auto')
```

A detailed information about initializing the Ray runtime you can find in starting ray page.

3.9.5 Usage example

In example below we train XGBoost model using the Iris Dataset and get prediction on the same data. All processing will be in a *single node* mode.

```
from sklearn import datasets

import ray
ray.init() # Start the Ray runtime for single-node

import modin.pandas as pd
import modin.experimental.xgboost as xgb

# Load iris dataset from sklearn
iris = datasets.load_iris()

# Create Modin DataFrames
X = pd.DataFrame(iris.data)
y = pd.DataFrame(iris.target)

# Create DMatrix
dtrain = xgb.DMatrix(X, y)
dtest = xgb.DMatrix(X, y)

# Set training parameters
xgb_params = {
    "eta": 0.3,
    "max_depth": 3,
    "objective": "multi:softprob",
    "num_class": 3,
    "eval_metric": "mlogloss",
}
steps = 20

# Create dict for evaluation results
evals_result = dict()
```

(continues on next page)

(continued from previous page)

```
# Run training
model = xgb.train(
    xgb_params,
    dtrain,
    steps,
    evals=[(dtrain, "train")],
    evals_result=evals_result
)

# Print evaluation results
print(f'Evals results:\n{evals_result}')

# Predict results
prediction = model.predict(dtest)

# Print prediction results
print(f'Prediction results:\n{prediction}')
```

3.10 Modin in the Cloud

Modin implements functionality that allows to transfer computing to the cloud with minimal effort. Please note that this feature is experimental and behavior or interfaces could be changed.

3.10.1 Prerequisites

Sign up with a cloud provider and get credentials file. Note that we supported only AWS currently, more are planned. (AWS credentials file format)

3.10.2 Setup environment

```
pip install modin[remote]
```

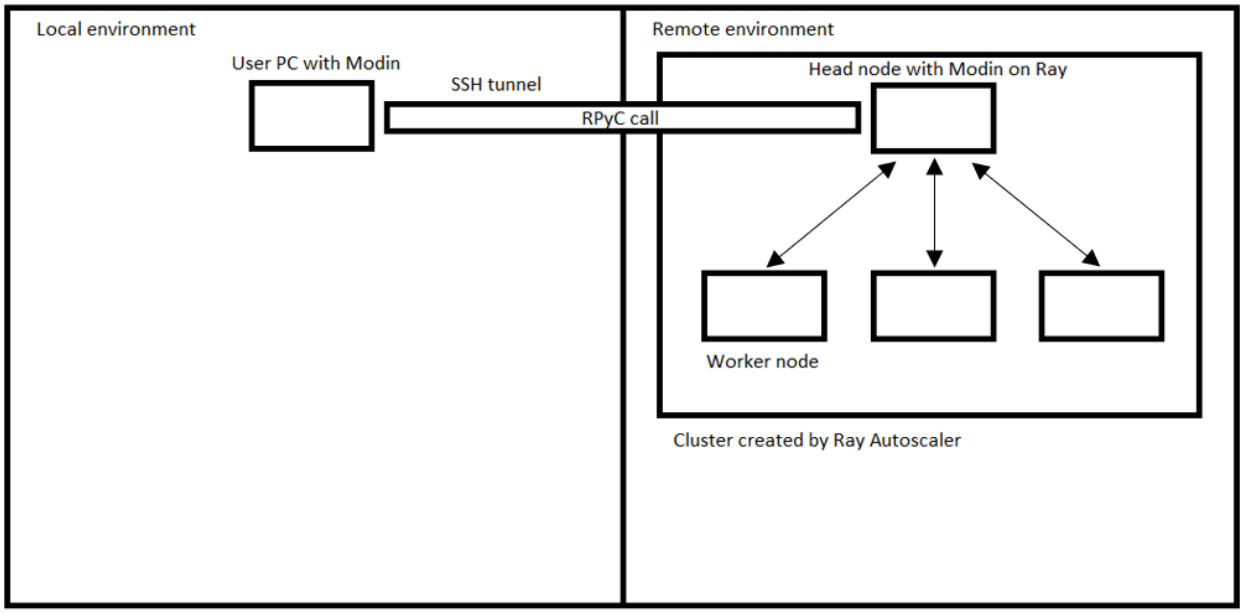
This command install the following dependencies:

- RPyC - allows to perform remote procedure calls.
- Cloudpickle - allows pickling of functions and classes, which is used in our distributed runtime.
- Boto3 - allows to create and setup AWS cloud machines. Optional library for Ray Autoscaler.

Notes:

- It also needs Ray Autoscaler component, which is implicitly installed with Ray (note that Ray from `conda` is now missing that component!). More information in Ray docs.

3.10.3 Architecture



Notes:

- To get maximum performance, you need to try to reduce the amount of data transferred between local and remote environments as much as possible.
- To ensure correct operation, it is necessary to ensure the equivalence of versions of all Python libraries (including the interpreter) in the local and remote environments.

3.10.4 Public interface

exception modin.experimental.cloud.**CannotDestroyCluster**(**args*, *cause: Optional[BaseException] = None*, *traceback: Optional[str] = None*, ***kw*)

Raised when cluster cannot be destroyed in the cloud

exception modin.experimental.cloud.**CannotSpawnCluster**(**args*, *cause: Optional[BaseException] = None*, *traceback: Optional[str] = None*, ***kw*)

Raised when cluster cannot be spawned in the cloud

exception modin.experimental.cloud.**ClusterError**(**args*, *cause: Optional[BaseException] = None*, *traceback: Optional[str] = None*, ***kw*)

Generic cluster operating exception

modin.experimental.cloud.**create_cluster**(*provider: Union[modin.experimental.cloud.cluster.Provider, str]*, *credentials: Optional[str] = None*, *region: Optional[str] = None*, *zone: Optional[str] = None*, *image: Optional[str] = None*, *project_name: Optional[str] = None*, *cluster_name: str = 'modin-cluster'*, *workers: int = 4*, *head_node: Optional[str] = None*, *worker_node: Optional[str] = None*, *add_conda_packages: Optional[list] = None*, *cluster_type: str = 'rayscale'*) → modin.experimental.cloud.cluster.BaseCluster

Creates an instance of a cluster with desired characteristics in a cloud. Upon entering a context via with statement

Modin will redirect its work to the remote cluster. Spawned cluster can be destroyed manually, or it will be destroyed when the program exits.

Parameters

- **provider** (*str or instance of Provider class*) – Specify the name of the provider to use or a Provider object. If Provider object is given, then credentials, region and zone are ignored.
- **credentials** (*str, optional*) – Path to the file which holds credentials used by given cloud provider. If not specified, cloud provider will use its default means of finding credentials on the system.
- **region** (*str, optional*) – Region in the cloud where to spawn the cluster. If omitted a default for given provider will be taken.
- **zone** (*str, optional*) – Availability zone (part of region) where to spawn the cluster. If omitted a default for given provider and region will be taken.
- **image** (*str, optional*) – Image to use for spawning head and worker nodes. If omitted a default for given provider will be taken.
- **project_name** (*str, optional*) – Project name to assign to the cluster in cloud, for easier manual tracking.
- **cluster_name** (*str, optional*) – Name to be given to the cluster. To spawn multiple clusters in single region and zone use different names.
- **workers** (*int, optional*) – How many worker nodes to spawn in the cluster. Head node is not counted for here.
- **head_node** (*str, optional*) – What machine type to use for head node in the cluster.
- **worker_node** (*str, optional*) – What machine type to use for worker nodes in the cluster.
- **add_conda_packages** (*list, optional*) – Custom conda packages for remote environments. By default remote modin version is the same as local version.
- **cluster_type** (*str, optional*) – How to spawn the cluster. Currently spawning by Ray autoscaler ("rayscale" for general and "omnisci" for Omnisci-based) is supported

Returns The object that knows how to destroy the cluster and how to activate it as remote context. Note that by default spawning and destroying of the cluster happens in the background, as it's usually a rather lengthy process.

Return type BaseCluster descendant

Notes

Cluster computation actually can work when proxies are required to access the cloud. You should set normal "http_proxy"/"https_proxy" variables for HTTP/HTTPS proxies and set "MODIN_SOCKS_PROXY" variable for SOCKS proxy before calling the function.

Using SOCKS proxy requires Ray newer than 0.8.6, which might need to be installed manually.

modin.experimental.cloud.**get_connection**()

Returns an RPyC connection object to execute Python code remotely on the active cluster.

3.10.5 Usage examples

```
"""
This is a very basic sample script for running things remotely.
It requires `aws_credentials` file to be present in current working directory.
On credentials file format see https://docs.aws.amazon.com/cli/latest/userguide/cli-
↪configure-files.html#cli-configure-files-where
"""
import logging
import modin.pandas as pd
from modin.experimental.cloud import cluster
# set up verbose logging so Ray autoscaler would print a lot of things
# and we'll see that stuff is alive and kicking
logging.basicConfig(format="%(asctime)s %(message)s")
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
example_cluster = cluster.create("aws", "aws_credentials")
with example_cluster:
    remote_df = pd.DataFrame([1, 2, 3, 4])
    print(len(remote_df))  # len() is executed remotely
```

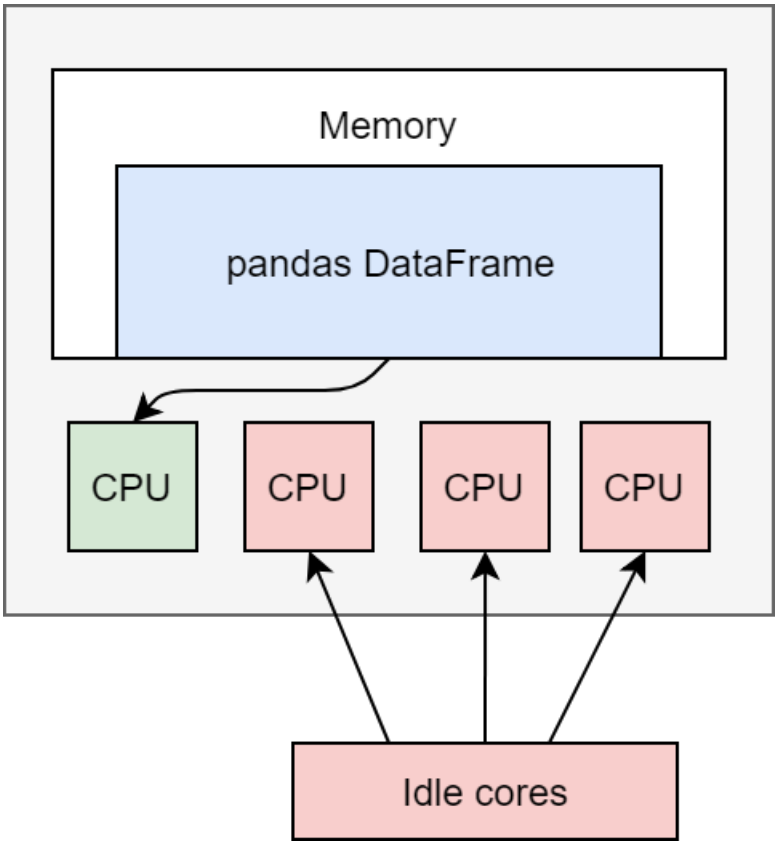
Some more examples can be found in `examples/cluster` folder.

3.11 Modin vs. pandas

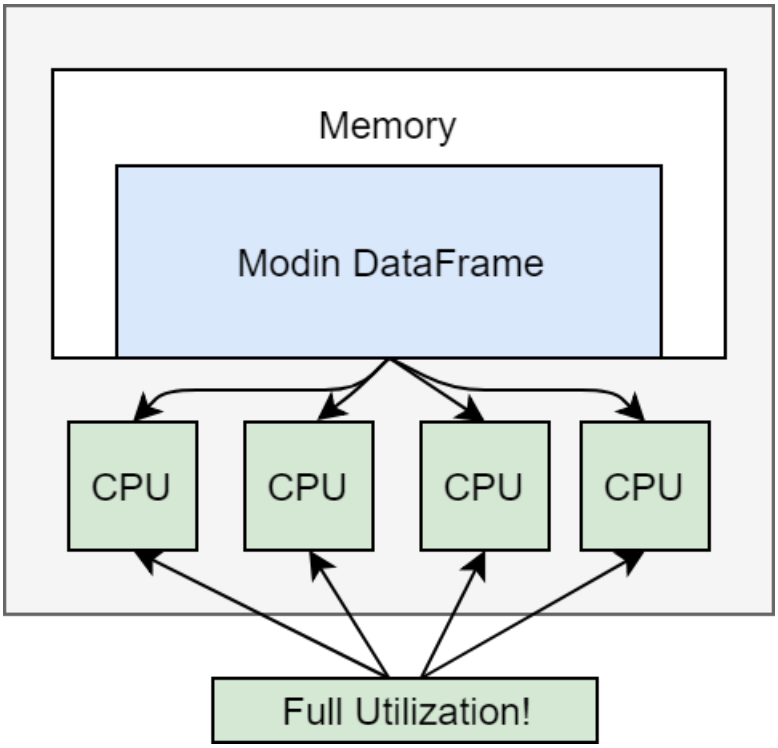
Modin exposes the pandas API through `modin.pandas`, but it does not inherit the same pitfalls and design decisions that make it difficult to scale. This page will discuss how Modin's dataframe implementation differs from pandas, and how Modin scales pandas.

3.11.1 Scalablity of implementation

The pandas implementation is inherently single-threaded. This means that only one of your CPU cores can be utilized at any given time. In a laptop, it would look something like this with pandas:

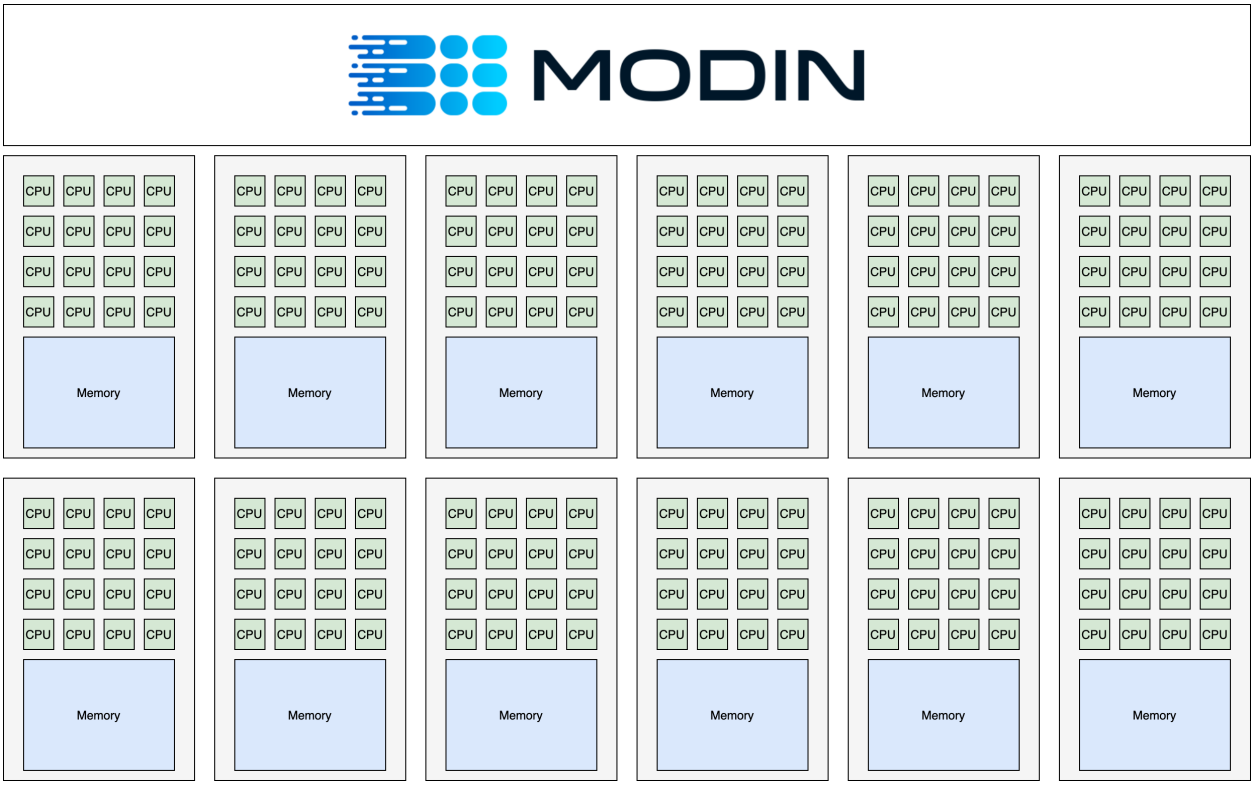


However, Modin's implementation enables you to use all of the cores on your machine, or all of the cores in an entire cluster. On a laptop, it will look something like this:



The additional utilization leads to improved performance, however if you want to scale to an entire cluster, Modin

suddenly looks something like this:



Modin is able to efficiently make use of all of the hardware available to it!

3.11.2 Memory usage and immutability

The pandas API contains many cases of "inplace" updates, which are known to be controversial. This is due in part to the way pandas manages memory: the user may think they are saving memory, but pandas is usually copying the data whether an operation was inplace or not.

Modin allows for inplace semantics, but the underlying data structures within Modin's implementation are immutable, unlike pandas. This immutability gives Modin the ability to internally chain operators and better manage memory layouts, because they will not be changed. This leads to improvements over pandas in memory usage in many common cases, due to the ability to share common memory blocks among all dataframes.

Modin provides the inplace semantics by having a mutable pointer to the immutable internal Modin dataframe. This pointer can change, but the underlying data cannot, so when an inplace update is triggered, Modin will treat it as if it were not inplace and just update the pointer to the resulting Modin dataframe.

3.11.3 API vs implementation

It is well known that the pandas API contains many duplicate ways of performing the same operation. Modin instead enforces that any one behavior have one and only one implementation internally. This guarantee enables Modin to focus on and optimize a smaller code footprint while still guaranteeing that it covers the entire pandas API. Modin has an internal algebra, which is roughly 15 operators, narrowed down from the original >200 that exist in pandas. The algebra is grounded in both practical and theoretical work. Learn more in our VLDB 2020 paper. More information about this algebra can be found in the *System Architecture* documentation.

3.12 Modin vs. Dask Dataframe

Dask's Dataframe is effectively a meta-frame, partitioning and scheduling many smaller `pandas.DataFrame` objects. The Dask DataFrame does not implement the entire pandas API, and it isn't trying to. See this explained in the Dask DataFrame documentation.

The TL;DR is that Modin's API is identical to pandas, whereas Dask's is not. Note: The projects are fundamentally different in their aims, so a fair comparison is challenging.

3.12.1 API

The API of Modin and Dask are different in several ways, explained here.

Dask DataFrame

Dask is currently missing multiple APIs from pandas that Modin has implemented. Of note: Dask does not implement `iloc`, `MultiIndex`, `apply(axis=0)`, `quantile` (approximate quantile is available), `median`, and more. Some of these APIs cannot be implemented efficiently or at all given the architecture design tradeoffs made in Dask's implementation, and others simply require engineering effort. `iloc`, for example, can be implemented, but it would be inefficient, and `apply(axis=0)` cannot be implemented at all in Dask's architecture.

Dask DataFrames API is also different from the pandas API in that it is lazy and needs `.compute()` calls to materialize the DataFrame. This makes the API less convenient but allows Dask to do certain query optimizations/rearrangement, which can give speedups in certain situations. Several additional APIs exist in the Dask DataFrame API that expose internal state about how the data is chunked and other data layout details, and ways to manipulate that state.

Semantically, Dask sorts the `index`, which does not allow for user-specified order. In Dask's case, this was done for optimization purposes, to speed up other computations which involve the row index.

Modin

Modin is targeted toward parallelizing the entire pandas API, without exception. As the pandas API continues to evolve, so will Modin's pandas API. Modin is intended to be used as a drop-in replacement for pandas, such that even if the API is not yet parallelized, it still works by falling back to running pandas. One of the key features of being a drop-in replacement is that not only will it work for existing code, if a user wishing to go back to running pandas directly, they may at no cost. There's no lock-in: Modin notebooks can be converted to and from pandas as the user prefers.

In the long-term, Modin is planned to become a data science framework that supports all popular APIs (SQL, pandas, etc.) with the same underlying execution.

3.12.2 Architecture

The differences in Modin and Dask's architectures are explained in this section.

Dask DataFrame

Dask DataFrame uses row-based partitioning, similar to Spark. This can be seen in their documentation. They also have a custom index object for indexing into the object, which is not pandas compatible. Dask DataFrame seems to treat operations on the DataFrame as MapReduce operations, which is a good paradigm for the subset of the pandas API they have chosen to implement, but makes certain operations impossible. Dask Dataframe is also lazy and places a lot of partitioning responsibility on the user.

Modin

Modin's partition is much more flexible, so the system can scale in both directions and have finer grained partitioning. This is explained at a high level in Modin's documentation. Because we have this finer grained control over the partitioning, we can support a number of operations that are very challenging in MapReduce systems (e.g. transpose, median, quantile). This flexibility in partitioning also gives Modin tremendous power to implement efficient straggler mitigation and improvements in utilization over the entire cluster.

Modin is also architected to run on a variety of systems. The goal here is that users can take the same notebook to different clusters or different environments and it will still just work, run on what you have! Modin does support running on Dask's compute engine in addition to Ray. The architecture of Modin is extremely modular, we are able to add different execution engines or compile to different memory formats because of this modularity. Modin can run on a Dask cluster in the same way that Dask Dataframe can, but they will still be different in all of the ways described above.

Modin's implementation is grounded in theory, which is what enables us to implement the entire pandas API.

3.13 Modin vs. Koalas and Spark

Coming Soon...

3.14 Supported APIs and Defaulting to pandas

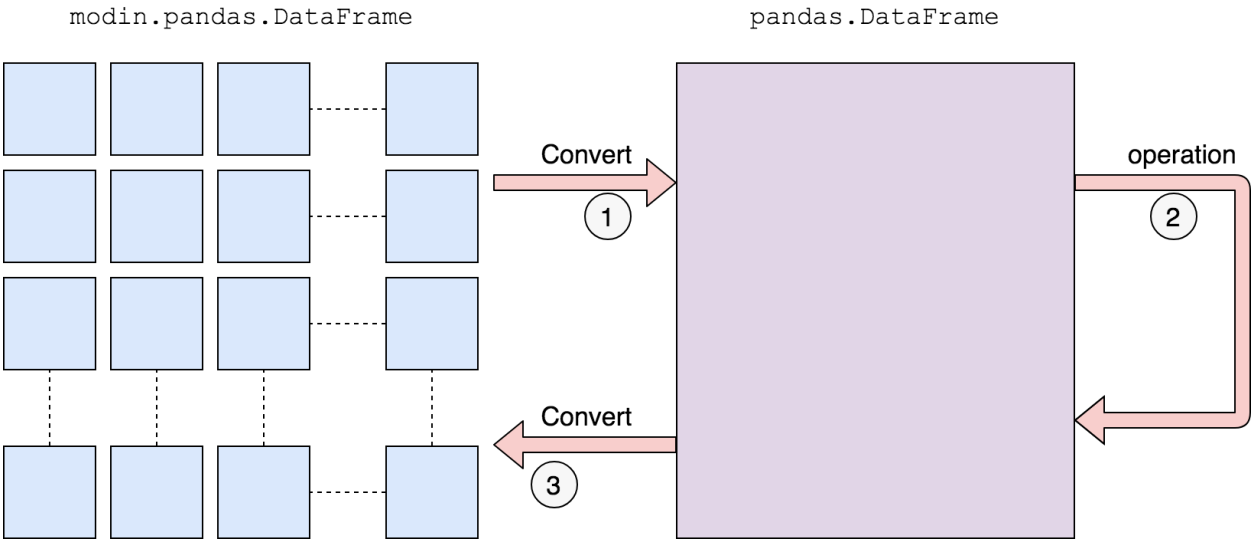
For your convenience, we have compiled a list of currently implemented APIs and methods available in Modin. This documentation is updated as new methods and APIs are merged into the master branch, and not necessarily correct as of the most recent release. In order to install the latest version of Modin, follow the directions found on the *installation page*.

3.14.1 Questions on implementation details

If you have a question about the implementation details or would like more information about an API or method in Modin, please contact the Modin developer mailing list.

3.14.2 Defaulting to pandas

The remaining unimplemented methods default to pandas. This allows users to continue using Modin even though their workloads contain functions not yet implemented in Modin. Here is a diagram of how we convert to pandas and perform the operation:



We first convert to a pandas DataFrame, then perform the operation. There is a performance penalty for going from a partitioned Modin DataFrame to pandas because of the communication cost and single-threaded nature of pandas. Once the pandas operation has completed, we convert the DataFrame back into a partitioned Modin DataFrame. This way, operations performed after something defaults to pandas will be optimized with Modin.

The exact methods we have implemented are listed in the respective subsections:

- *DataFrame*
- *Series*
- *utilities*
- *I/O*

We have taken a community-driven approach to implementing new methods. We did a study on pandas usage to learn what the most-used APIs are. Modin currently supports **93%** of the pandas API based on our study of pandas usage, and we are actively expanding the API.

3.15 pd.DataFrame supported APIs

The following table lists both implemented and not implemented methods. If you have need of an operation that is listed as not implemented, feel free to open an issue on the GitHub repository, or give a thumbs up to already created issues. Contributions are also welcome!

The following table is structured as follows: The first column contains the method name. The second column is a flag for whether or not there is an implementation in Modin for the method in the left column. `Y` stands for yes, `N` stands for no, `P` stands for partial (meaning some parameters may not be supported yet), and `D` stands for default to pandas.

DataFrame method	pandas Doc link	Implemented? (Y/N/P/D)	Notes for Current implementation
`T`	T	Y	
`abs`	abs	Y	
`add`	add	Y	Shuffles data in operations between DataFrames
`add_prefix`	add_prefix	Y	
`add_suffix`	add_suffix	Y	
`agg` / `aggregate`	agg / aggregate	P	• Dictionary `func` parameter defaults to pandas • Numpy operations default to pandas
`align`	align	D	
`all`	all	Y	
`any`	any	Y	
`append`	append	Y	
`apply`	apply	Y	See `agg`
`applymap`	applymap	Y	
`as_blocks`	as_blocks	D	Becomes a non-parallel object
`as_matrix`	as_matrix	D	Becomes a non-parallel object
`asfreq`	asfreq	D	
`asof`	asof	Y	
`assign`	assign	Y	
`astype`	astype	Y	
`at`	at	Y	
`at_time`	at_time	Y	
`axes`	axes	Y	
`between_time`	between_time	Y	
`bfill`	bfill	Y	
`blocks`	blocks	D	
`bool`	bool	Y	
`boxplot`	boxplot	D	
`clip`	clip	Y	
`clip_lower`	clip_lower	Y	
`clip_upper`	clip_upper	Y	
`combine`	combine	Y	

continues on next page

Table 1 – continued from previous page

combine_first	combine_first	Y	
compare	compare	Y	
copy	copy	Y	
corr	corr	Y	Correlation floating point precision may slightly differ from pandas. For now pearson method is available only. For other methods defaults to pandas.
corrwith	corrwith	D	
count	count	Y	
cov	cov	Y	Covariance floating point precision may slightly differ from pandas.
cummax	cummax	Y	
cummin	cummin	Y	
cumprod	cumprod	Y	
cumsum	cumsum	Y	
describe	describe	Y	
diff	diff	Y	
div	div	Y	See add
divide	divide	Y	See add
dot	dot	Y	
drop	drop	Y	
droplevel	droplevel	Y	
drop_duplicates	drop_duplicates	D	
dropna	dropna	Y	
dtypes	dtypes	Y	
duplicated	duplicated	Y	
empty	empty	Y	
eq	eq	Y	See add
equals	equals	Y	Requires shuffle, can be further optimized
eval	eval	Y	
ewm	ewm	D	
expanding	expanding	D	
explode	explode	Y	
ffill	ffill	Y	
fillna	fillna	P	value parameter of type DataFrame defaults to pandas
filter	filter	Y	
first	first	Y	
first_valid_index	first_valid_index	Y	
floordiv	floordiv	Y	See add
from_dict	from_dict	D	
from_items	from_items	Y	
from_records	from_records	D	
ftypes	ftypes	Y	
ge	ge	Y	See add
get	get	Y	

continues on next page

Table 1 – continued from previous page

groupby	groupby	Y	Not yet optimized for all operations
gt	gt	Y	See add
head	head	Y	
hist	hist	D	
iat	iat	Y	
idxmax	idxmax	Y	
idxmin	idxmin	Y	
iloc	iloc	Y	
infer_objects	infer_objects	D	
info	info	Y	
insert	insert	Y	
interpolate	interpolate	D	
isin	isin	Y	
isna	isna	Y	
isnull	isnull	Y	
items	items	Y	
iteritems	iteritems	P	Modin does not parallelize iteration in Python
iterrows	iterrows	P	Modin does not parallelize iteration in Python
itertuples	itertuples	P	Modin does not parallelize iteration in Python
join	join	P	When on is set to right or outer it defaults to pandas
keys	keys	Y	
kurt	kurt	Y	
kurtosis	kurtosis	Y	
last	last	Y	
last_valid_index	last_valid_index	Y	
le	le	Y	See add
loc	loc	Y	We do not support: boolean array, callable
lookup	lookup	D	
lt	lt	Y	See add
mad	mad	Y	
mask	mask	D	
max	max	Y	
mean	mean	P	Modin defaults to pandas if given the level param.
median	median	P	Modin defaults to pandas if given the level param.
melt	melt	Y	
memory_usage	memory_usage	Y	

continues on next page

Table 1 – continued from previous page

merge	merge	P	Implemented the following cases: left_index=True and right_index=True, how=left and how=inner for all values of parameters except left_index=True and right_index=False or left_index=False and right_index=True. Defaults to pandas otherwise.
min	min	Y	
mod	mod	Y	
mode	mode	Y	
mul	mul	Y	See add
multiply	multiply	Y	See add
ndim	ndim	Y	
ne	ne	Y	See add
nlargest	nlargest	Y	
notna	notna	Y	
notnull	notnull	Y	
nsmallest	nsmallest	Y	
nunique	nunique	Y	
pct_change	pct_change	D	
pipe	pipe	Y	
pivot	pivot	Y	
pivot_table	pivot_table	Y	
plot	plot	D	
pop	pop	Y	
pow	pow	Y	See add
prod	prod	Y	
product	product	Y	
quantile	quantile	Y	
query	query	P	Local variables not yet supported
radd	radd	Y	See add
rank	rank	Y	
rdiv	rdiv	Y	See add
reindex	reindex	Y	Shuffles data
reindex_like	reindex_like	D	
rename	rename	Y	
rename_axis	rename_axis	Y	
reorder_levels	reorder_levels	Y	
replace	replace	Y	
resample	resample	Y	
reset_index	reset_index	Y	
rfloordiv	rfloordiv	Y	See add
rmod	rmod	Y	See add
rmul	rmul	Y	See add

continues on next page

Table 1 – continued from previous page

rolling	rolling	Y	
round	round	Y	
rpow	rpow	Y	See add
rsub	rsub	Y	See add
rtruediv	rtruediv	Y	See add
sample	sample	Y	
select_dtypes	select_dtypes	Y	
sem	sem	P	Modin defaults to pandas if given the level param.
set_axis	set_axis	Y	
set_index	set_index	Y	
shape	shape	Y	
shift	shift	Y	
size	size	Y	
skew	skew	P	Modin defaults to pandas if given the level param.
slice_shift	slice_shift	Y	
sort_index	sort_index	Y	
sort_values	sort_values	Y	Shuffles data
sparse	sparse	N	
squeeze	squeeze	Y	
stack	stack	Y	
std	std	P	Modin defaults to pandas if given the level param.
style	style	D	
sub	sub	Y	See add
subtract	subtract	Y	See add
sum	sum	Y	
swapaxes	swapaxes	Y	
swaplevel	swaplevel	Y	
tail	tail	Y	
take	take	Y	
to_clipboard	to_clipboard	D	
to_csv	to_csv	Y	
to_dense	to_dense	D	
to_dict	to_dict	D	
to_excel	to_excel	D	
to_feather	to_feather	D	
to_gbq	to_gbq	D	
to_hdf	to_hdf	D	
to_html	to_html	D	
to_json	to_json	D	
to_latex	to_latex	D	
to_msgpack	to_msgpack	D	
to_parquet	to_parquet	D	
to_period	to_period	D	
to_pickle	to_pickle	D	Experimental implementation: to_pickle_distributed
to_records	to_records	D	

continues on next page

Table 1 – continued from previous page

to_sparse	to_sparse	D	
to_sql	to_sql	Y	
to_stata	to_stata	D	
to_string	to_string	D	
to_timestamp	to_timestamp	D	
to_xarray	to_xarray	D	
transform	transform	Y	
transpose	transpose	Y	
truediv	truediv	Y	See add
truncate	truncate	Y	
tshift	tshift	Y	
tz_convert	tz_convert	Y	
tz_localize	tz_localize	Y	
unstack	unstack	Y	
update	update	Y	
values	values	Y	
value_counts	value_counts	D	
var	var	P	Modin defaults to pandas if given the level param.
where	where	Y	

3.16 pd.Series supported APIs

The following table lists both implemented and not implemented methods. If you have need of an operation that is listed as not implemented, feel free to open an issue on the GitHub repository, or give a thumbs up to already created issues. Contributions are also welcome!

The following table is structured as follows: The first column contains the method name. The second column is a flag for whether or not there is an implementation in Modin for the method in the left column. Y stands for yes, N stands for no, P stands for partial (meaning some parameters may not be supported yet), and D stands for default to pandas. To learn more about the implementations that default to pandas, see the related section on *Defaulting to pandas*.

Series method	Modin Implementation? (Y/N/P/D)	Notes for Current implementation
abs	Y	
add	Y	
add_prefix	Y	
add_suffix	Y	
agg	Y	
aggregate	Y	
align	D	
all	Y	
any	Y	
append	Y	
apply	Y	
argmax	Y	
argmin	Y	
argsort	D	
array	D	
as_blocks	D	

Table 2 – continued from previous page

as_matrix	Y	
asfreq	D	
asobject	D	
asof	Y	
astype	Y	
at	Y	
at_time	Y	
autocorr	Y	
axes	Y	
base	D	
between	D	
between_time	Y	
bfill	Y	
blocks	D	
bool	Y	
cat	D	
clip	Y	
clip_lower	Y	
clip_upper	Y	
combine	Y	
combine_first	Y	
compare	Y	
compress	D	
copy	Y	
corr	Y	Correlation floating point precision may slightly differ from pa
count	Y	
cov	Y	Covariance floating point precision may slightly differ from pa
cummax	Y	
cummin	Y	
cumprod	Y	
cumsum	Y	
data	D	
describe	Y	
diff	Y	
div	Y	
divide	Y	
divmod	Y	
dot	Y	
drop	Y	
drop_duplicates	Y	
droplevel	Y	
dropna	Y	
dt	Y	
dtype	Y	
dtypes	Y	
duplicated	Y	
empty	Y	
eq	Y	
equals	Y	
ewm	D	

Table 2 – continued from previous page

expanding	D	
explode	Y	
factorize	D	
ffill	Y	
fillna	Y	
filter	Y	
first	Y	
first_valid_index	Y	
flags	D	
floordiv	Y	
from_array	D	
ftype	Y	
ftypes	Y	
ge	Y	
get	Y	
get_dtype_counts	Y	
get_ftype_counts	Y	
get_value	D	
get_values	D	
groupby	D	
gt	Y	
hasnans	Y	
head	Y	
hist	D	
iat	Y	
idxmax	Y	
idxmin	Y	
iloc	Y	
imag	D	
index	Y	
infer_objects	D	
interpolate	D	
is_monotonic	Y	
is_monotonic_decreasing	Y	
is_monotonic_increasing	Y	
is_unique	Y	
isin	Y	
isna	Y	
isnull	Y	
item	Y	
items	Y	
itemsize	D	
iteritems	Y	
keys	Y	
kurt	Y	
kurtosis	Y	
last	Y	
last_valid_index	Y	
le	Y	
loc	Y	

Table 2 – continued from previous page

lt	Y	
mad	Y	
map	Y	
mask	D	
max	Y	
mean	P	Modin defaults to pandas if given the `level` param.
median	P	Modin defaults to pandas if given the `level` param.
memory_usage	Y	
min	Y	
mod	Y	
mode	Y	
mul	Y	
multiply	Y	
name	Y	
nbytes	D	
ndim	Y	
ne	Y	
nlargest	Y	
nonzero	Y	
notna	Y	
notnull	Y	
nsmallest	Y	
nunique	Y	
pct_change	D	
pipe	Y	
plot	D	
pop	Y	
pow	Y	
prod	Y	
product	Y	
ptp	D	
put	D	
quantile	Y	
radd	Y	
rank	Y	
ravel	Y	
rdiv	Y	
rdivmod	Y	
real	D	
reindex	Y	
reindex_like	Y	
rename	Y	
rename_axis	Y	
reorder_levels	D	
repeat	D	
replace	Y	
resample	Y	
reset_index	Y	
rfloordiv	Y	
rmod	Y	

Table 2 – continued from previous page

rmul	Y	
rolling	Y	
round	Y	
rpow	Y	
rsub	Y	
rtruediv	Y	
sample	Y	
searchsorted	Y	
sem	P	Modin defaults to pandas if given the `level` param.
set_axis	Y	
set_value	D	
shape	Y	
shift	Y	
size	Y	
skew	P	Modin defaults to pandas if given the `level` param.
slice_shift	Y	
sort_index	Y	
sort_values	Y	
sparse	Y	
squeeze	Y	
std	P	Modin defaults to pandas if given the `level` param.
str	Y	
strides	D	
sub	Y	
subtract	Y	
sum	Y	
swapaxes	Y	
swaplevel	Y	
tail	Y	
take	Y	
to_clipboard	D	
to_csv	D	
to_dense	D	
to_dict	D	
to_excel	D	
to_frame	Y	
to_hdf	D	
to_json	D	
to_latex	D	
to_list	D	
to_msgpack	D	
to_numpy	D	
to_period	D	
to_pickle	D	
to_sparse	D	
to_sql	Y	
to_string	D	
to_timestamp	D	
to_xarray	D	
tolist	D	

Table 2 – continued from previous page

transform	Y	
transpose	Y	
truediv	Y	
truncate	Y	
tshift	Y	
tz_convert	Y	
tz_localize	Y	
unique	Y	
unstack	Y	
update	Y	
valid	D	
value_counts	Y	The indices order of resulting object may differ from pandas.
values	Y	
var	P	Modin defaults to pandas if given the `level` param.
view	D	
where	Y	

3.17 pandas Utilities Supported

If you `import modin.pandas as pd` the following operations are available from `pd.<op>`, e.g. `pd.concat`. If you do not see an operation that pandas enables and would like to request it, feel free to open an issue. Make sure you tell us your primary use-case so we can make it happen faster!

The following table is structured as follows: The first column contains the method name. The second column is a flag for whether or not there is an implementation in Modin for the method in the left column. `Y` stands for yes, `N` stands for no, `P` stands for partial (meaning some parameters may not be supported yet), and `D` stands for default to pandas.

Utility method	Modin Implementation? (Y/N/P/D)	Notes for Current implementation
pd.concat	Y	
pd.eval	Y	
pd.unique	Y	
pd.value_counts	Y	The indices order of resulting object may differ from pandas.
pd.cut	D	
pd.to_numeric	D	
pd.factorize	D	
pd.qcut	D	
pd.match	D	
pd.to_datetime	D	
pd.get_dummies	Y	
pd.date_range	D	
pd.bdate_range	D	
pd.to_timedelta	D	
pd.options	Y	
pd.datetime	D	

3.17.1 Other objects & structures

This list is a list of objects not currently distributed by Modin. All of these objects are compatible with the distributed components of Modin. If you are interested in contributing a distributed version of any of these objects, feel free to open a pull request.

- Panel
- Index
- MultiIndex
- CategoricalIndex
- DatetimeIndex
- Timedelta
- Timestamp
- NaT
- PeriodIndex
- Categorical
- Interval
- UInt8Dtype
- UInt16Dtype
- UInt32Dtype
- UInt64Dtype
- SparseDtype
- Int8Dtype
- Int16Dtype
- Int32Dtype
- Int64Dtype
- CategoricalDtype
- DatetimeTZDtype
- IntervalDtype
- PeriodDtype
- RangeIndex
- Int64Index
- UInt64Index
- Float64Index
- TimedeltaIndex
- IntervalIndex
- IndexSlice
- TimeGrouper

- Grouper
- array
- Period
- DateOffset
- ExcelWriter
- SparseArray
- SparseSeries
- SparseDataFrame

3.18 pd.read_<file> and I/O APIs

A number of IO methods default to pandas. We have parallelized `read_csv` and `read_parquet`, though many of the remaining methods can be relatively easily parallelized. Some of the operations default to the pandas implementation, meaning it will read in serially as a single, non-distributed DataFrame and distribute it. Performance will be affected by this.

The following table is structured as follows: The first column contains the method name. The second column is a flag for whether or not there is an implementation in Modin for the method in the left column. `Y` stands for yes, `N` stands for no, `P` stands for partial (meaning some parameters may not be supported yet), and `D` stands for default to pandas.

IO method	Modin Implementation? (Y/N/P/D)	Notes for Current implementation
read_csv	Y	
read_table	Y	
read_parquet	Y	
read_json	P	Implemented for `lines=True`
read_html	D	
read_clipboard	D	
read_excel	D	
read_hdf	D	
read_feather	Y	
read_msgpack	D	
read_stata	D	
read_sas	D	
read_pickle	D	Experimental implementation: read_pickle_distributed
read_sql	Y	

3.19 Contributing

3.19.1 Getting Started

If you're interested in getting involved in the development of Modin, but aren't sure where start, take a look at the issues tagged Good first issue or Documentation. These are issues that would be good for getting familiar with the codebase and better understanding some of the more complex components of the architecture. There is documentation here about the *architecture* that you will want to review in order to get started.

Also, feel free to join the discussions on the developer mailing list.

If you want a quick guide to getting your development enviroment setup, please use the contributing instructions on GitHub.

3.19.2 Certificate of Origin

To keep a clear track of who did what, we use a *sign-off* procedure (same requirements for using the signed-off-by process as the Linux kernel has https://www.kernel.org/doc/html/v4.17/process/submitting-patches.html) on patches or pull requests that are being sent. The sign-off is a simple line at the end of the explanation for the patch, which certifies that you wrote it or otherwise have the right to pass it on as an open-source patch. The rules are pretty simple: if you can certify the below:

CERTIFICATE OF ORIGIN V 1.1

"By making a contribution to this project, I certify that:

1.) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or 2.) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or 3.) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it. 4.) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved."

```
This is my commit message

Signed-off-by: Awesome Developer <developer@example.org>
```

Code without a proper signoff cannot be merged into the master branch. Note: You must use your real name (sorry, no pseudonyms or anonymous contributions.)

The text can either be manually added to your commit body, or you can add either `-s` or `--signoff` to your usual `git commit` commands:

```
git commit --signoff
git commit -s
```

This will use your default git configuration which is found in .git/config. To change this, you can use the following commands:

```
git config --global user.name "Awesome Developer"
git config --global user.email "awesome.developer.@example.org"
```

If you have authored a commit that is missing the signed-off-by line, you can amend your commits and push them to GitHub.

```
git commit --amend --signoff
```

If you've pushed your changes to GitHub already you'll need to force push your branch after this with `git push -f`.

3.19.3 Commit Message formatting

To ensure that all commit messages in the master branch follow a specific format, we enforce that all commit messages must follow the following format:

```
FEAT-#9999: Add `DataFrame.rolling` functionality, to enable rolling window operations
```

The `FEAT` component represents the type of commit. This component of the commit message can be one of the following:

- FEAT: A new feature that is added
- DOCS: Documentation improvements or updates
- FIX: A bugfix contribution
- REFACTOR: Moving or removing code without change in functionality
- TEST: Test updates or improvements

The `#9999` component of the commit message should be the issue number in the Modin GitHub issue tracker: https://github.com/modin-project/modin/issues. This is important because it links commits to their issues.

The commit message should follow a colon (:) and be descriptive and succinct.

3.19.4 General Rules for committers

- Try to write a PR name as descriptive as possible.
- Try to keep PRs as small as possible. One PR should be making one semantically atomic change.
- Don't merge your own PRs even if you are technically able to do it.

3.19.5 Development Dependencies

We recommend doing development in a virtualenv or conda environment, though this decision is ultimately yours. You will want to run the following in order to install all of the required dependencies for running the tests and formatting the code:

```
conda env create --file environment-dev.yml
# or
pip install -r requirements-dev.txt
```

3.19.6 Code Formatting and Lint

We use black for code formatting. Before you submit a pull request, please make sure that you run the following from the project root:

```
black modin/ asv_bench/benchmarks scripts/doc_checker.py
```

We also use flake8 to check linting errors. Running the following from the project root will ensure that it passes the lint checks on Github Actions:

```
flake8 modin/ asv_bench/benchmarks scripts/doc_checker.py
```

We test that this has been run on our Github Actions test suite. If you do this and find that the tests are still failing, try updating your version of black and flake8.

3.19.7 Adding a test

If you find yourself fixing a bug or adding a new feature, don't forget to add a test to the test suite to verify its correctness! More on testing and the layout of the tests can be found in our testing documentation. We ask that you follow the existing structure of the tests for ease of maintenance.

3.19.8 Running the tests

To run the entire test suite, run the following from the project root:

```
pytest modin/pandas/test
```

The test suite is very large, and may take a long time if you run every test. If you've only modified a small amount of code, it may be sufficient to run a single test or some subset of the test suite. In order to run a specific test run:

```
pytest modin/pandas/test::test_new_functionality
```

The entire test suite is automatically run for each pull request.

3.19.9 Performance measurement

We use Asv tool for performance tracking of various Modin functionality. The results can be viewed here: Asv dashboard.

More information can be found in the Asv readme.

3.19.10 Building documentation

To build the documentation, please follow the steps below from the project root:

```
cd docs
pip install -r requirements-doc.txt
sphinx-build -b html . build
```

To visualize the documentation locally, run the following from *build* folder:

```
python -m http.server <port>
# python -m http.server 1234
```

then open the browser at *0.0.0.0:<port>* (e.g. *0.0.0.0:1234*).

3.19.11 Contributing a new execution framework or in-memory format

If you are interested in contributing support for a new execution framework or in-memory format, please make sure you understand the *architecture* of Modin.

The best place to start the discussion for adding a new execution framework or in-memory format is the developer mailing list.

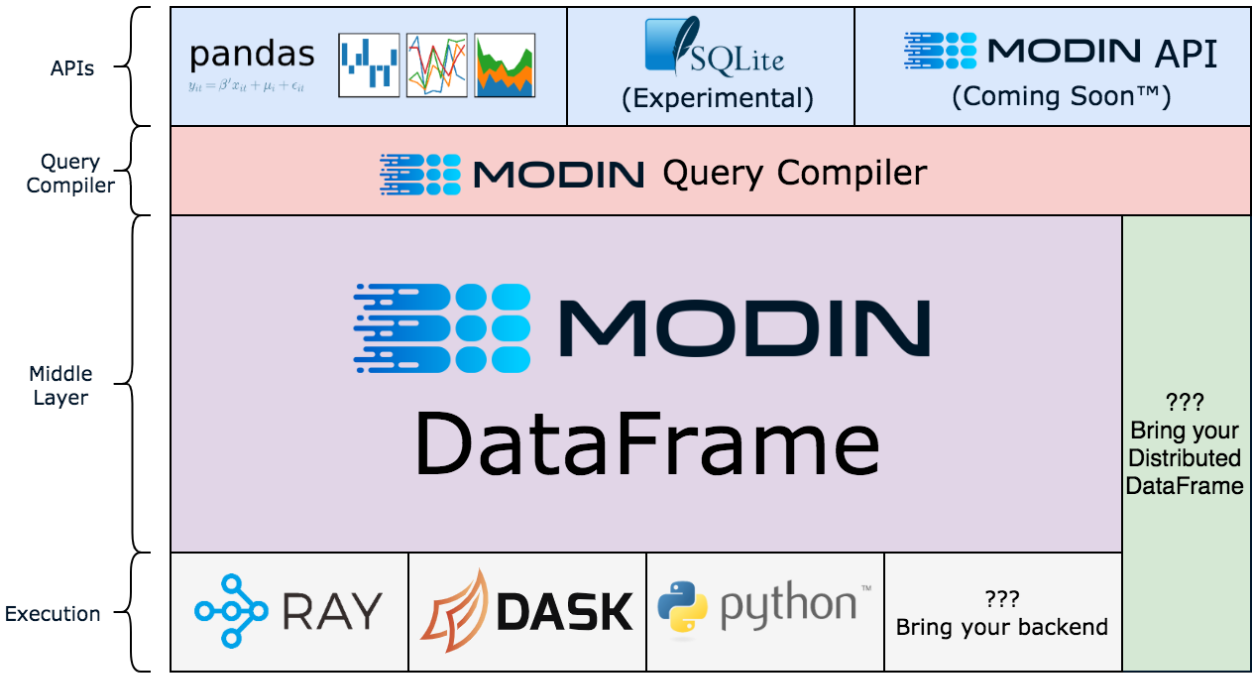
More docs on this coming soon...

3.20 System Architecture

In this section, we will lay out the overall system architecture for Modin, as well as go into detail about the component design, implementation and other important details. This document also contains important reference information for those interested in contributing new functionality, bugfixes and enhancements.

3.20.1 High-Level Architectural View

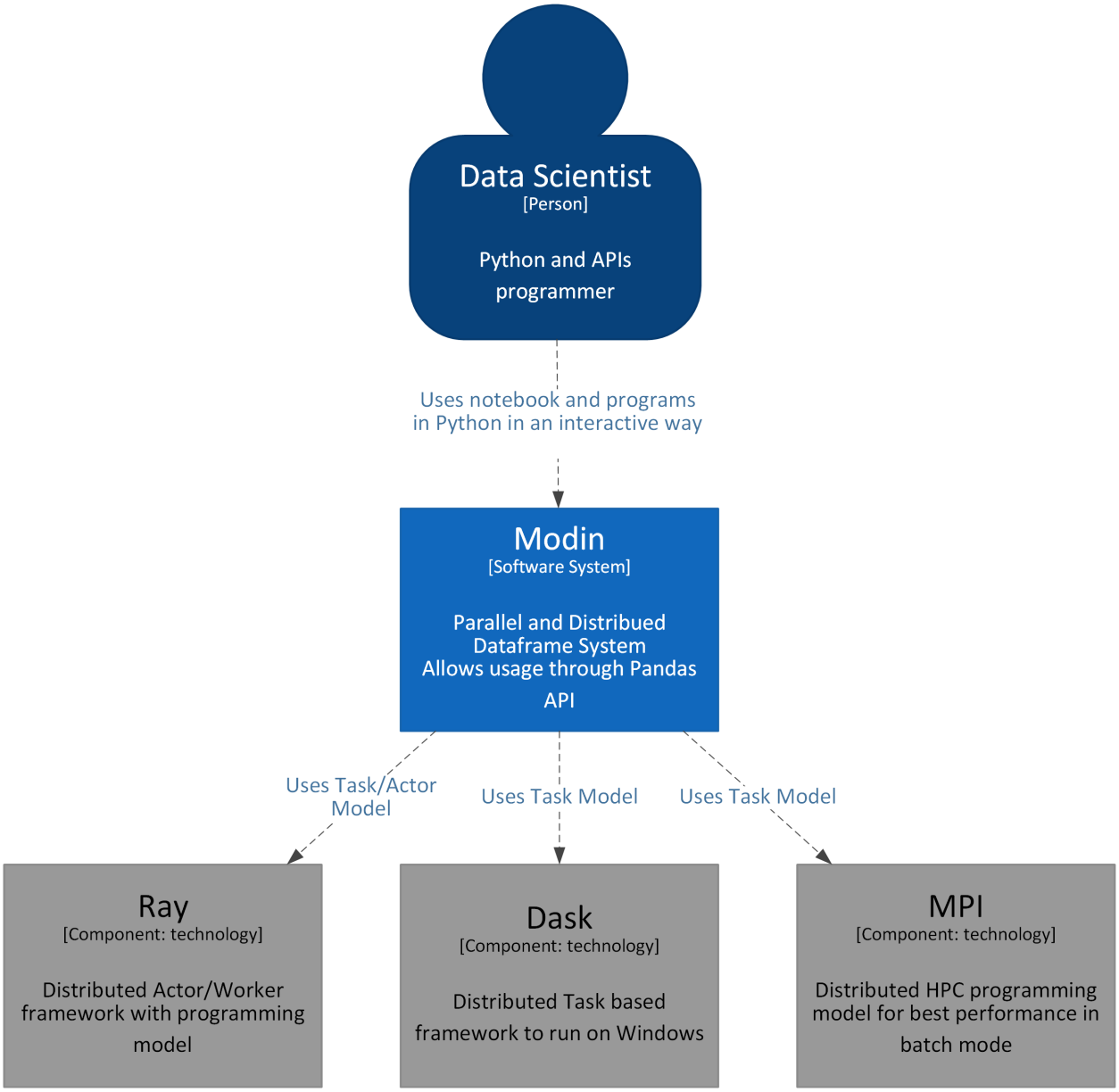
The diagram below outlines the general layered view to the components of Modin with a short description of each major section of the documentation following.



Modin is logically separated into different layers that represent the hierarchy of a typical Database Management System. Abstracting out each component allows us to individually optimize and swap out components without affecting the rest of the system. We can implement, for example, new compute kernels that are optimized for a certain type of data and can simply plug it in to the existing infrastructure by implementing a small interface. It can still be distributed by our choice of compute engine with the logic internally.

3.20.2 System View

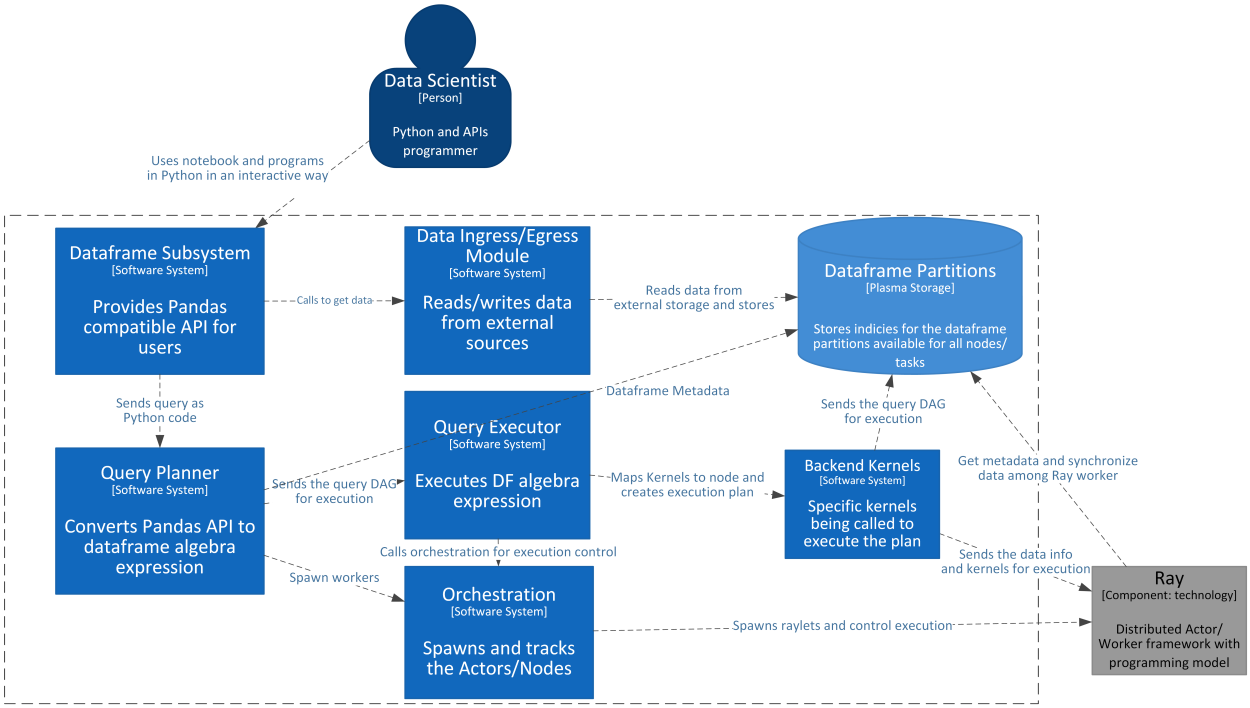
If we look to the overall class structure of the Modin system from very top, it will look to something like this:



The user - Data Scientist interacts with the Modin system by sending interactive or batch commands through API and Modin executes them using various execution engines: Ray, Dask and MPI are currently supported.

3.20.3 Subsystem/Container View

If we click down to the next level of details we will see that inside Modin the layered architecture is implemented using several interacting components:



For the simplicity the other execution systems - Dask and MPI are omitted and only Ray execution is shown.

- Dataframe subsystem is the backbone of the dataframe holding and query compilation. It is responsible for dispatching the ingress/egress to the appropriate module, getting the Pandas API and calling the query compiler to convert calls to the internal intermediate Dataframe Algebra.
- Data Ingress/Egress Module is working in conjunction with Dataframe and Partitions subsystem to read data split into partitions and send data into the appropriate node for storing.
- Query Planner is subsystem that translates the Pandas API to intermediate Dataframe Algebra representation DAG and performs an initial set of optimizations.
- Query Executor is responsible for getting the Dataframe Algebra DAG, performing further optimizations based on a selected storage format and mapping or compiling the Dataframe Algebra DAG to and actual execution sequence.
- Storage formats module is responsible for mapping the abstract operation to an actual executor call, e.g. Pandas, PyArrow, custom format.
- Orchestration subsystem is responsible for spawning and controlling the actual execution environment for the selected execution. It spawns the actual nodes, fires up the execution environment, e.g. Ray, monitors the state of executors and provides telemetry

3.20.4 Component View

Modin PandasDataframe Objects

- *PandasDataframe* is the class conforming to Dataframe Algebra.
- *PandasDataframePartition* implements `Partition` interface holding `pandas.DataFrame`.
- *PandasDataframeAxisPartition* is a joined group of `PandasDataframePartition`-s along some axis (either rows or labels)
- *PandasDataframePartitionManager* is the manager that implements the primitives used for Dataframe Algebra operations over `PandasDataframePartition`-s

PandasDataframe

The class is base for any frame class of `pandas` storage format and serves as the intermediate level between `pandas` query compiler and conforming partition manager. All queries formed at the query compiler layer are ingested by this class and then conveyed jointly with the stored partitions into the partition manager for processing. Direct partitions manipulation by this class is prohibited except cases if an operation is stricly private or protected and called inside of the class only. The class provides significantly reduced set of operations that fit plenty of pandas operations.

Main tasks of *PandasDataframe* are storage of partitions, manipulation with labels of axes and providing set of methods to perform operations on the internal data.

As mentioned above, `PandasDataframe` shouldn't work with stored partitions directly and the responsibility for modifying partitions array has to lay on *PandasDataframePartitionManager*. For example, method *broadcast_apply_full_axis()* redirects applying function to `broadcast_axis_partitions()` method.

`Modin PandasDataframe` can be created from `pandas.DataFrame`, `pyarrow.Table` (methods *from_pandas()*, *from_arrow()* are used respectively). Also, `PandasDataframe` can be converted to `np.array`, `pandas.DataFrame` (methods *to_numpy()*, *to_pandas()* are used respectively).

Manipulation with labels of axes happens using internal methods for changing labels on the new, adding prefixes/suffixes etc.

Public API

class modin.core.dataframe.pandas.dataframe.dataframe.**PandasDataframe**(*partitions*, *index*, *columns*, *row_lengths=None*, *column_widths=None*, *dtypes=None*)

An abstract class that represents the parent class for any pandas storage format dataframe class.

This class provides interfaces to run operations on dataframe partitions.

Parameters

- **partitions** (*np.ndarray*) – A 2D NumPy array of partitions.
- **index** (*sequence*) – The index for the dataframe. Converted to a `pandas.Index`.
- **columns** (*sequence*) – The columns object for the dataframe. Converted to a `pandas.Index`.
- **row_lengths** (*list, optional*) – The length of each partition in the rows. The "height" of each of the block partitions. Is computed if not provided.

- **column_widths** (*list, optional*) – The width of each partition in the columns. The "width" of each of the block partitions. Is computed if not provided.
- **dtypes** (*pandas.Series, optional*) – The data types for the dataframe columns.

add_prefix(*prefix*, *axis*)
Add a prefix to the current row or column labels.

Parameters

- **prefix** (*str*) – The prefix to add.
- **axis** (*int*) – The axis to update.

Returns A new dataframe with the updated labels.

Return type *PandasDataframe*

add_suffix(*suffix*, *axis*)
Add a suffix to the current row or column labels.

Parameters

- **suffix** (*str*) – The suffix to add.
- **axis** (*int*) – The axis to update.

Returns A new dataframe with the updated labels.

Return type *PandasDataframe*

apply_full_axis(*axis*, *func*, *new_index=None*, *new_columns=None*, *dtypes=None*)
Perform a function across an entire axis.

Parameters

- **axis** (*{0, 1}*) – The axis to apply over (0 - rows, 1 - columns).
- **func** (*callable*) – The function to apply.
- **new_index** (*list-like, optional*) – The index of the result. We may know this in advance, and if not provided it must be computed.
- **new_columns** (*list-like, optional*) – The columns of the result. We may know this in advance, and if not provided it must be computed.
- **dtypes** (*list-like, optional*) – The data types of the result. This is an optimization because there are functions that always result in a particular data type, and allows us to avoid (re)computing it.

Returns A new dataframe.

Return type *PandasDataframe*

Notes

The data shape may change as a result of the function.

apply_full_axis_select_indices(*axis*, *func*, *apply_indices=None*, *numeric_indices=None*, *new_index=None*, *new_columns=None*, *keep_remaining=False*)
Apply a function across an entire axis for a subset of the data.

Parameters

- **axis** (*int*) – The axis to apply over.

- **func** (*callable*) – The function to apply.
- **apply_indices** (*list-like, default: None*) – The labels to apply over.
- **numeric_indices** (*list-like, default: None*) – The indices to apply over.
- **new_index** (*list-like, optional*) – The index of the result. We may know this in advance, and if not provided it must be computed.
- **new_columns** (*list-like, optional*) – The columns of the result. We may know this in advance, and if not provided it must be computed.
- **keep_remaining** (*boolean, default: False*) – Whether or not to drop the data that is not computed over.

Returns A new dataframe.

Return type *PandasDataframe*

apply_select_indices(*axis*, *func*, *apply_indices=None*, *row_indices=None*, *col_indices=None*, *new_index=None*, *new_columns=None*, *keep_remaining=False*, *item_to_distribute=None*)

Apply a function for a subset of the data.

Parameters

- **axis** (*{0, 1}*) – The axis to apply over.
- **func** (*callable*) – The function to apply.
- **apply_indices** (*list-like, default: None*) – The labels to apply over. Must be given if axis is provided.
- **row_indices** (*list-like, default: None*) – The row indices to apply over. Must be provided with *col_indices* to apply over both axes.
- **col_indices** (*list-like, default: None*) – The column indices to apply over. Must be provided with *row_indices* to apply over both axes.
- **new_index** (*list-like, optional*) – The index of the result. We may know this in advance, and if not provided it must be computed.
- **new_columns** (*list-like, optional*) – The columns of the result. We may know this in advance, and if not provided it must be computed.
- **keep_remaining** (*boolean, default: False*) – Whether or not to drop the data that is not computed over.
- **item_to_distribute** (*(optional)*) – The item to split up so it can be applied over both axes.

Returns A new dataframe.

Return type *PandasDataframe*

astype(*col_dtypes*)

Convert the columns dtypes to given dtypes.

Parameters **col_dtypes** (*dictionary of {col: dtype,...}*) – Where col is the column name and dtype is a NumPy dtype.

Returns Dataframe with updated dtypes.

Return type BaseDataFrame

property axes
Get index and columns that can be accessed with an *axis* integer.

Returns List with two values: index and columns.

Return type list

binary_op(*op*, *right_frame*, *join_type='outer'*)
Perform an operation that requires joining with another Modin DataFrame.

Parameters

- **op** (*callable*) – Function to apply after the join.
- **right_frame** (PandasDataframe) – Modin DataFrame to join with.
- **join_type** (*str, default: "outer"*) – Type of join to apply.

Returns New Modin DataFrame.

Return type *PandasDataframe*

broadcast_apply(*axis*, *func*, *other*, *join_type='left'*, *preserve_labels=True*, *dtypes=None*)
Broadcast axis partitions of *other* to partitions of *self* and apply a function.

Parameters

- **axis** (*{0, 1}*) – Axis to broadcast over.
- **func** (*callable*) – Function to apply.
- **other** (PandasDataframe) – Modin DataFrame to broadcast.
- **join_type** (*str, default: "left"*) – Type of join to apply.
- **preserve_labels** (*bool, default: True*) – Whether keep labels from *self* Modin DataFrame or not.
- **dtypes** (*"copy" or None, default: None*) – Whether keep old dtypes or infer new dtypes from data.

Returns New Modin DataFrame.

Return type *PandasDataframe*

broadcast_apply_full_axis(*axis*, *func*, *other*, *new_index=None*, *new_columns=None*, *apply_indices=None*, *enumerate_partitions=False*, *dtypes=None*)
Broadcast partitions of *other* Modin DataFrame and apply a function along full axis.

Parameters

- **axis** (*{0, 1}*) – Axis to apply over (0 - rows, 1 - columns).
- **func** (*callable*) – Function to apply.
- **other** (PandasDataframe *or list*) – Modin DataFrame(s) to broadcast.
- **new_index** (*list-like, optional*) – Index of the result. We may know this in advance, and if not provided it must be computed.
- **new_columns** (*list-like, optional*) – Columns of the result. We may know this in advance, and if not provided it must be computed.
- **apply_indices** (*list-like, default: None*) – Indices of *axis ^ 1* to apply function over.
- **enumerate_partitions** (*bool, default: False*) – Whether pass partition index into applied *func* or not. Note that *func* must be able to obtain *partition_idx* kwarg.

- **dtypes** (*list-like, default: None*) – Data types of the result. This is an optimization because there are functions that always result in a particular data type, and allows us to avoid (re)computing it.

Returns New Modin DataFrame.

Return type *PandasDataframe*

broadcast_apply_select_indices(*axis*, *func*, *other*, *apply_indices=None*, *numeric_indices=None*, *keep_remaining=False*, *broadcast_all=True*, *new_index=None*, *new_columns=None*)

Apply a function to select indices at specified axis and broadcast partitions of *other* Modin DataFrame.

Parameters

- **axis** (*{0, 1}*) – Axis to apply function along.
- **func** (*callable*) – Function to apply.
- **other** (PandasDataframe) – Partitions of which should be broadcasted.
- **apply_indices** (*list, default: None*) – List of labels to apply (if *numeric_indices* are not specified).
- **numeric_indices** (*list, default: None*) – Numeric indices to apply (if *apply_indices* are not specified).
- **keep_remaining** (*bool, default: False*) – Whether drop the data that is not computed over or not.
- **broadcast_all** (*bool, default: True*) – Whether broadcast the whole axis of right frame to every partition or just a subset of it.
- **new_index** (*pandas.Index, optional*) – Index of the result. We may know this in advance, and if not provided it must be computed.
- **new_columns** (*pandas.Index, optional*) – Columns of the result. We may know this in advance, and if not provided it must be computed.

Returns New Modin DataFrame.

Return type *PandasDataframe*

property columns

Get the columns from the cache object.

Returns An index object containing the column labels.

Return type pandas.Index

classmethod combine_dtypes(*list_of_dtypes*, *column_names*)

Describe how data types should be combined when they do not match.

Parameters

- **list_of_dtypes** (*list*) – A list of pandas Series with the data types.
- **column_names** (*list*) – The names of the columns that the data types map to.

Returns A pandas Series containing the finalized data types.

Return type pandas.Series

concat(*axis*, *others*, *how*, *sort*)

Concatenate *self* with one or more other Modin DataFrames.

Parameters

- **axis** (*{0, 1}*) – Axis to concatenate over.
- **others** (*list*) – List of Modin DataFrames to concatenate with.
- **how** (*str*) – Type of join to use for the axis.
- **sort** (*bool*) – Whether sort the result or not.

Returns New Modin DataFrame.

Return type *PandasDataframe*

copy()
Copy this object.

Returns A copied version of this object.

Return type *PandasDataframe*

property dtypes
Compute the data types if they are not cached.

Returns A pandas Series containing the data types for this dataframe.

Return type pandas.Series

explode(*axis*, *func*)
Explode list-like entries along an entire axis.

Parameters

- **axis** (*int*) – The axis specifying how to explode. If axis=1, explode according to columns.
- **func** (*callable*) – The function to use to explode a single element.

Returns A new filtered dataframe.

Return type PandasFrame

filter_full_axis(*axis*, *func*)
Filter data based on the function provided along an entire axis.

Parameters

- **axis** (*int*) – The axis to filter over.
- **func** (*callable*) – The function to use for the filter. This function should filter the data itself.

Returns A new filtered dataframe.

Return type *PandasDataframe*

finalize()
Perform all deferred calls on partitions.

This makes *self* Modin Dataframe independent of a history of queries that were used to build it.

fold(*axis*, *func*)
Perform a function across an entire axis.

Parameters

- **axis** (*int*) – The axis to apply over.
- **func** (*callable*) – The function to apply.

Returns A new dataframe.

Return type *PandasDataframe*

Notes

The data shape is not changed (length and width of the table).

fold_reduce(*axis*, *func*)
Apply function that reduces Frame Manager to series but requires knowledge of full axis.

Parameters

- **axis** (*{0, 1}*) – The axis to apply the function to (0 - index, 1 - columns).
- **func** (*callable*) – The function to reduce the Manager by. This function takes in a Manager.

Returns Modin series (1xN frame) containing the reduced data.

Return type *PandasDataframe*

classmethod from_arrow(*at*)
Create a Modin DataFrame from an Arrow Table.

Parameters **at** (*pyarrow.table*) – Arrow Table.

Returns New Modin DataFrame.

Return type *PandasDataframe*

from_labels() → *modin.core.dataframe.pandas.dataframe.dataframe.PandasDataframe*
Convert the row labels to a column of data, inserted at the first position.

Gives result by similar way as *pandas.DataFrame.reset_index*. Each level of *self.index* will be added as separate column of data.

Returns A PandasDataframe with new columns from index labels.

Return type *PandasDataframe*

classmethod from_pandas(*df*)
Create a Modin DataFrame from a pandas DataFrame.

Parameters **df** (*pandas.DataFrame*) – A pandas DataFrame.

Returns New Modin DataFrame.

Return type *PandasDataframe*

groupby_reduce(*axis*, *by*, *map_func*, *reduce_func*, *new_index=None*, *new_columns=None*, *apply_indices=None*)
Groupby another Modin DataFrame dataframe and aggregate the result.

Parameters

- **axis** (*{0, 1}*) – Axis to groupby and aggregate over.
- **by** (*PandasDataframe or None*) – A Modin DataFrame to group by.
- **map_func** (*callable*) – Map component of the aggregation.
- **reduce_func** (*callable*) – Reduce component of the aggregation.
- **new_index** (*pandas.Index, optional*) – Index of the result. We may know this in advance, and if not provided it must be computed.

- **new_columns** (*pandas.Index, optional*) – Columns of the result. We may know this in advance, and if not provided it must be computed.
- **apply_indices** (*list-like, default: None*) – Indices of *axis ^ 1* to apply groupby over.

Returns New Modin DataFrame.

Return type *PandasDataframe*

property index
Get the index from the cache object.

Returns An index object containing the row labels.

Return type pandas.Index

map(*func*, *dtypes=None*)
Perform a function that maps across the entire dataset.

Parameters

- **func** (*callable*) – The function to apply.
- **dtypes** (*dtypes of the result, default: None*) – The data types for the result. This is an optimization because there are functions that always result in a particular data type, and this allows us to avoid (re)computing it.

Returns A new dataframe.

Return type *PandasDataframe*

map_reduce(*axis*, *map_func*, *reduce_func=None*)
Apply function that will reduce the data to a pandas Series.

Parameters

- **axis** (*{0, 1}*) – 0 for columns and 1 for rows.
- **map_func** (*callable*) – Callable function to map the dataframe.
- **reduce_func** (*callable, default: None*) – Callable function to reduce the dataframe. If none, then apply map_func twice.

Returns A new dataframe.

Return type *PandasDataframe*

mask(*row_indices=None*, *row_numeric_idx=None*, *col_indices=None*, *col_numeric_idx=None*)
Lazily select columns or rows from given indices.

Parameters

- **row_indices** (*list of hashable, optional*) – The row labels to extract.
- **row_numeric_idx** (*list-like of ints, optional*) – The row indices to extract.
- **col_indices** (*list of hashable, optional*) – The column labels to extract.
- **col_numeric_idx** (*list-like of ints, optional*) – The column indices to extract.

Returns A new PandasDataframe from the mask provided.

Return type *PandasDataframe*

Notes

If both *row_indices* and *row_numeric_idx* are set, *row_indices* will be used. The same rule applied to *col_indices* and *col_numeric_idx*.

numeric_columns(*include_bool=True*)
Return the names of numeric columns in the frame.

Parameters **include_bool** (*bool, default: True*) – Whether to consider boolean columns as numeric.

Returns List of column names.

Return type list

synchronize_labels(*axis=None*)
Synchronize labels by applying the index object for specific *axis* to the *self._partitions* lazily.

Adds *set_axis* function to call-queue of each partition from *self._partitions* to apply new axis.

Parameters **axis** (*int, default: None*) – The axis to apply to. If it's None applies to both axes.

to_labels(*column_list: List[Hashable]*) → *modin.core.dataframe.pandas.dataframe.dataframe.PandasDataframe*
Move one or more columns into the row labels. Previous labels are dropped.

Parameters **column_list** (*list of hashable*) – The list of column names to place as the new row labels.

Returns A new PandasDataframe that has the updated labels.

Return type *PandasDataframe*

to_numpy(***kwargs*)
Convert this Modin DataFrame to a NumPy array.

Parameters ****kwargs** (*dict*) – Additional keyword arguments to be passed in *to_numpy*.

Returns

Return type np.ndarray

to_pandas()
Convert this Modin DataFrame to a pandas DataFrame.

Returns

Return type pandas.DataFrame

transpose()
Transpose the index and columns of this Modin DataFrame.

Reflect this Modin DataFrame over its main diagonal by writing rows as columns and vice-versa.

Returns New Modin DataFrame.

Return type *PandasDataframe*

PandasDataframePartition

The class is base for any partition class of `pandas` storage format and serves as the last level on which operations that were conveyed from the partition manager are being performed on an individual block partition.

The class provides an API that has to be overridden by child classes in order to manipulate on data and metadata they store.

The public API exposed by the children of this class is used in *PandasDataframePartitionManager*.

The objects wrapped by the child classes are treated as immutable by `PandasDataframePartitionManager` subclasses and no logic for updating inplace.

Public API

class modin.core.dataframe.pandas.partitioning.partition.**PandasDataframePartition**

An abstract class that is base for any partition class of `pandas` storage format.

The class providing an API that has to be overridden by child classes.

add_to_apply_calls(*func*, **args*, ***kwargs*)

Add a function to the call queue.

Parameters

- **func** (*callable*) – Function to be added to the call queue.
- ***args** (*iterable*) – Additional positional arguments to be passed in *func*.
- ****kwargs** (*dict*) – Additional keyword arguments to be passed in *func*.

Returns New *PandasDataframePartition* object with the function added to the call queue.

Return type *PandasDataframePartition*

Notes

This function will be executed when *apply* is called. It will be executed in the order inserted; apply's func operates the last and return.

apply(*func*, **args*, ***kwargs*)

Apply a function to the object wrapped by this partition.

Parameters

- **func** (*callable*) – Function to apply.
- ***args** (*iterable*) – Additional positional arguments to be passed in *func*.
- ****kwargs** (*dict*) – Additional keyword arguments to be passed in *func*.

Returns New *PandasDataframePartition* object.

Return type *PandasDataframePartition*

Notes

It is up to the implementation how *kwargs* are handled. They are an important part of many implementations. As of right now, they are not serialized.

drain_call_queue()
Execute all operations stored in the call queue on the object wrapped by this partition.

classmethod empty()
Create a new partition that wraps an empty pandas DataFrame.

Returns New *PandasDataframePartition* object.

Return type *PandasDataframePartition*

get()
Get the object wrapped by this partition.

Returns The object that was wrapped by this partition.

Return type object

Notes

This is the opposite of the classmethod *put*. E.g. if you assign *x = PandasDataframePartition.put(1)*, *x.get()* should always return 1.

length()
Get the length of the object wrapped by this partition.

Returns The length of the object.

Return type int

mask(*row_indices*, *col_indices*)
Lazily create a mask that extracts the indices provided.

Parameters

- **row_indices** (*list-like, slice or label*) – The indices for the rows to extract.
- **col_indices** (*list-like, slice or label*) – The indices for the columns to extract.

Returns New *PandasDataframePartition* object.

Return type *PandasDataframePartition*

classmethod preprocess_func(*func*)
Preprocess a function before an *apply* call.

Parameters **func** (*callable*) – Function to preprocess.

Returns An object that can be accepted by *apply*.

Return type callable

Notes

This is a classmethod because the definition of how to preprocess should be class-wide. Also, we may want to use this before we deploy a preprocessed function to multiple *PandasDataframePartition* objects.

classmethod put(*obj*)
Put an object into a store and wrap it with partition object.

Parameters **obj** (*object*) – An object to be put.

Returns New *PandasDataframePartition* object.

Return type *PandasDataframePartition*

to_numpy(***kwargs*)
Convert the object wrapped by this partition to a NumPy array.

Parameters ****kwargs** (*dict*) – Additional keyword arguments to be passed in *to_numpy*.

Returns

Return type np.ndarray

Notes

If the underlying object is a pandas DataFrame, this will return a 2D NumPy array.

to_pandas()
Convert the object wrapped by this partition to a pandas DataFrame.

Returns

Return type pandas.DataFrame

Notes

If the underlying object is a pandas DataFrame, this will likely only need to call *get*.

wait()
Wait for completion of computations on the object wrapped by the partition.

width()
Get the width of the object wrapped by the partition.

Returns The width of the object.

Return type int

PandasDataframeAxisPartition

The class is base for any axis partition class of `pandas` storage format.

Subclasses must implement `list_of_blocks` which represents data wrapped by the *PandasDataframePartition* objects and creates something interpretable as a `pandas.DataFrame`.

See *PandasOnRayDataframeAxisPartition* for an example on how to override/use this class when the implementation needs to be augmented.

The *PandasDataframeAxisPartition* object has an invariant that requires that this object is never returned from a function. It assumes that there will always be `PandasDataframeAxisPartition` object stored and structures itself accordingly.

Public API

class modin.core.dataframe.pandas.partitioning.axis_partition.**PandasDataframeAxisPartition**

An abstract class is created to simplify and consolidate the code for axis partition that run pandas.

Because much of the code is similar, this allows us to reuse this code.

apply(*func*, *num_splits=None*, *other_axis_partition=None*, *maintain_partitioning=True*, ***kwargs*)

Apply a function to this axis partition along full axis.

Parameters

- **func** (*callable*) – The function to apply.
- **num_splits** (*int, default: None*) – The number of times to split the result object.
- **other_axis_partition** (PandasDataframeAxisPartition, *default: None*) – Another *PandasDataframeAxisPartition* object to be applied to func. This is for operations that are between two data sets.
- **maintain_partitioning** (*bool, default: True*) – Whether to keep the partitioning in the same orientation as it was previously or not. This is important because we may be operating on an individual AxisPartition and not touching the rest. In this case, we have to return the partitioning to its previous orientation (the lengths will remain the same). This is ignored between two axis partitions.
- ****kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.

Returns A list of *PandasDataframePartition* objects.

Return type list

classmethod deploy_axis_func(*axis*, *func*, *num_splits*, *kwargs*, *maintain_partitioning*, **partitions*)

Deploy a function along a full axis.

Parameters

- **axis** (*{0, 1}*) – The axis to perform the function along.
- **func** (*callable*) – The function to perform.
- **num_splits** (*int*) – The number of splits to return (see *split_result_of_axis_func_pandas*).
- **kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.
- **maintain_partitioning** (*bool*) – If True, keep the old partitioning if possible. If False, create a new partition layout.
- ***partitions** (*iterable*) – All partitions that make up the full axis (row or column).

Returns A list of pandas DataFrames.

Return type list

classmethod deploy_func_between_two_axis_partitions(*axis*, *func*, *num_splits*, *len_of_left*, *other_shape*, *kwargs*, **partitions*)

Deploy a function along a full axis between two data sets.

Parameters

- **axis** (*{0, 1}*) – The axis to perform the function along.
- **func** (*callable*) – The function to perform.

- **num_splits** (*int*) – The number of splits to return (see *split_result_of_axis_func_pandas*).
- **len_of_left** (*int*) – The number of values in *partitions* that belong to the left data set.
- **other_shape** (*np.ndarray*) – The shape of right frame in terms of partitions, i.e. (other_shape[i-1], other_shape[i]) will indicate slice to restore i-1 axis partition.
- **kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.
- ***partitions** (*iterable*) – All partitions that make up the full axis (row or column) for both data sets.

Returns A list of pandas DataFrames.

Return type list

shuffle(*func*, *lengths*, ***kwargs*)

Shuffle the order of the data in this axis partition based on the *lengths*.

Parameters

- **func** (*callable*) – The function to apply before splitting.
- **lengths** (*list*) – The list of partition lengths to split the result into.
- ****kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.

Returns A list of *PandasDataframePartition* objects split by *lengths*.

Return type list

PandasDataframePartitionManager

The class is base for any partition manager class of `pandas` storage format and serves as intermediate level between *Modin PandasDataframe* and conforming *partition* class. The class is responsible for partitions manipulation and applying a function to individual partitions: block partitions, row partitions or column partitions, i.e. the class can form axis partitions from block partitions to apply a function if an operation requires access to an entire column or row. The class translates frame API into partition API and also can have some preprocessing operations depending on the partition type for improving performance (for example, *preprocess_func()*).

Main task of partition manager is to keep knowledge of how partitions are stored and managed internal to itself, so surrounding code could use it via lean enough API without worrying about implementation details.

Partition manager can apply user-passed (arbitrary) function in different modes:

- block-wise (apply a function to individual block partitions):
  - optinally accepting partition indices along each axis
  - optionally accepting an item to be split so parts of it would be sent to each partition
- along a full axis (apply a function to an entire column or row made up of block partitions when user function needs information about the whole axis)

It can also broadcast partitions from *right* to *left* when executing certain operations making *right* partitions available for functions executed where *left* live.

Partition manager also is used to create "logical" partitions, or *axis partitions* by joining existing partitions along specified axis (either rows or labels), and to concatenate different partition sets along given axis.

It also maintains mapping from "external" (end user-visible) indices along all axes to internal indices which are actually pairs of indices of partitions and indices inside the partitions, as well as manages conversion to numpy and pandas representations.

Public API

class modin.core.dataframe.pandas.partitioning.partition_manager.**PandasDataframePartitionManager**

Base class for managing the dataframe data layout and operators across the distribution of partitions.

Partition class is the class to use for storing each partition. Each partition must extend the *PandasDataframePartition* class.

classmethod apply_func_to_indices_both_axis(*partitions*, *func*, *row_partitions_list*, *col_partitions_list*, *item_to_distribute=None*, *row_lengths=None*, *col_widths=None*)

Apply a function along both axes.

Parameters

- **partitions** (*np.ndarray*) – The partitions to which the *func* will apply.
- **func** (*callable*) – The function to apply.
- **row_partitions_list** (*iterable of tuples*) –

  Iterable of tuples, containing 2 values:

  1. Integer row partition index.
  2. Internal row indexer of this partition.
- **col_partitions_list** (*iterable of tuples*) –

  Iterable of tuples, containing 2 values:

  1. Integer column partition index.
  2. Internal column indexer of this partition.
- **item_to_distribute** (*item, default: None*) – The item to split up so it can be applied over both axes.
- **row_lengths** (*list of ints, optional*) – Lengths of partitions for every row. If not specified this information is extracted from partitions itself.
- **col_widths** (*list of ints, optional*) – Widths of partitions for every column. If not specified this information is extracted from partitions itself.

Returns A NumPy array with partitions.

Return type np.ndarray

Notes

For your func to operate directly on the indices provided, it must use *row_internal_indices*, *col_internal_indices* as keyword arguments.

classmethod apply_func_to_select_indices(*axis*, *partitions*, *func*, *indices*, *keep_remaining=False*)

Apply a function to select indices.

Parameters

- **axis** (*{0, 1}*) – Axis to apply the *func* over.
- **partitions** (*np.ndarray*) – The partitions to which the *func* will apply.
- **func** (*callable*) – The function to apply to these indices of partitions.

- **indices** (*dict*) – The indices to apply the function to.
- **keep_remaining** (*bool, default: False*) – Whether or not to keep the other partitions. Some operations may want to drop the remaining partitions and keep only the results.

Returns A NumPy array with partitions.

Return type np.ndarray

Notes

Your internal function must take a kwarg *internal_indices* for this to work correctly. This prevents information leakage of the internal index to the external representation.

classmethod apply_func_to_select_indices_along_full_axis(*axis*, *partitions*, *func*, *indices*, *keep_remaining=False*)

Apply a function to a select subset of full columns/rows.

Parameters

- **axis** (*{0, 1}*) – The axis to apply the function over.
- **partitions** (*np.ndarray*) – The partitions to which the *func* will apply.
- **func** (*callable*) – The function to apply.
- **indices** (*list-like*) – The global indices to apply the func to.
- **keep_remaining** (*bool, default: False*) – Whether or not to keep the other partitions. Some operations may want to drop the remaining partitions and keep only the results.

Returns A NumPy array with partitions.

Return type np.ndarray

Notes

This should be used when you need to apply a function that relies on some global information for the entire column/row, but only need to apply a function to a subset. For your func to operate directly on the indices provided, it must use *internal_indices* as a keyword argument.

classmethod axis_partition(*partitions*, *axis*)

Logically partition along given axis (columns or rows).

Parameters

- **partitions** (*list-like*) – List of partitions to be combined.
- **axis** (*{0, 1}*) – 0 for column partitions, 1 for row partitions.

Returns A list of *BaseDataframeAxisPartition* objects.

Return type list

classmethod binary_operation(*axis*, *left*, *func*, *right*)

Apply a function that requires two PandasDataframe objects.

Parameters

- **axis** (*{0, 1}*) – The axis to apply the function over (0 - rows, 1 - columns).
- **left** (*np.ndarray*) – The partitions of left PandasDataframe.

- **func** (*callable*) – The function to apply.
- **right** (*np.ndarray*) – The partitions of right PandasDataframe.

Returns A NumPy array with new partitions.

Return type np.ndarray

classmethod broadcast_apply(*axis*, *apply_func*, *left*, *right*, *other_name='r'*)

Broadcast the *right* partitions to *left* and apply *apply_func* function.

Parameters

- **axis** (*{0, 1}*) – Axis to apply and broadcast over.
- **apply_func** (*callable*) – Function to apply.
- **left** (*NumPy 2D array*) – Left partitions.
- **right** (*NumPy 2D array*) – Right partitions.
- **other_name** (*str, default: "r"*) – Name of key-value argument for *apply_func* that is used to pass *right* to *apply_func*.

Returns An of partition objects.

Return type NumPy array

Notes

This will often be overridden by implementations. It materializes the entire partitions of the right and applies them to the left through *apply*.

classmethod broadcast_apply_select_indices(*axis*, *apply_func*, *left*, *right*, *left_indices*, *right_indices*, *keep_remaining=False*)

Broadcast the *right* partitions to *left* and apply *apply_func* to selected indices.

Parameters

- **axis** (*{0, 1}*) – Axis to apply and broadcast over.
- **apply_func** (*callable*) – Function to apply.
- **left** (*NumPy 2D array*) – Left partitions.
- **right** (*NumPy 2D array*) – Right partitions.
- **left_indices** (*list-like*) – Indices to apply function to.
- **right_indices** (*dictionary of indices of right partitions*) – Indices that you want to bring at specified left partition, for example dict {key: {key1: [0, 1], key2: [5]}} means that in left[key] you want to broadcast [right[key1], right[key2]] partitions and internal indices for *right* must be [[0, 1], [5]].
- **keep_remaining** (*bool, default: False*) – Whether or not to keep the other partitions. Some operations may want to drop the remaining partitions and keep only the results.

Returns An array of partition objects.

Return type NumPy array

Notes

Your internal function must take these kwargs: [*internal_indices*, *other*, *internal_other_indices*] to work correctly!

classmethod broadcast_axis_partitions(*axis*, *apply_func*, *left*, *right*, *keep_partitioning=False*, *apply_indices=None*, *enumerate_partitions=False*, *lengths=None*, ***kwargs*)

Broadcast the *right* partitions to *left* and apply *apply_func* along full *axis*.

Parameters

- **axis** (*{0, 1}*) – Axis to apply and broadcast over.
- **apply_func** (*callable*) – Function to apply.
- **left** (*NumPy 2D array*) – Left partitions.
- **right** (*NumPy 2D array*) – Right partitions.
- **keep_partitioning** (*boolean, default: False*) – The flag to keep partition boundaries for Modin Frame. Setting it to True disables shuffling data from one partition to another.
- **apply_indices** (*list of ints, default: None*) – Indices of *axis ^ 1* to apply function over.
- **enumerate_partitions** (*bool, default: False*) – Whether or not to pass partition index into *apply_func*. Note that *apply_func* must be able to accept *partition_idx* kwarg.
- **lengths** (*list of ints, default: None*) – The list of lengths to shuffle the object.
- ****kwargs** (*dict*) – Additional options that could be used by different engines.

Returns An array of partition objects.

Return type NumPy array

classmethod column_partitions(*partitions*)

Get the list of *BaseDataframeAxisPartition* objects representing column-wise paritions.

Parameters **partitions** (*list-like*) – List of (smaller) partitions to be combined to column-wise partitions.

Returns A list of *BaseDataframeAxisPartition* objects.

Return type list

Notes

Each value in this list will be an *BaseDataframeAxisPartition* object. *BaseDataframeAxisPartition* is located in *axis_partition.py*.

classmethod concat(*axis*, *left_parts*, *right_parts*)

Concatenate the blocks of partitions with another set of blocks.

Parameters

- **axis** (*int*) – The axis to concatenate to.
- **left_parts** (*np.ndarray*) – NumPy array of partitions to concatenate with.

- **right_parts** (*np.ndarray or list*) – NumPy array of partitions to be concatenated.

Returns A new NumPy array with concatenated partitions.

Return type np.ndarray

Notes

Assumes that the blocks are already the same shape on the dimension being concatenated. A ValueError will be thrown if this condition is not met.

classmethod concatenate(*dfs*)
Concatenate pandas DataFrames with saving 'category' dtype.

Parameters **dfs** (*list*) – List of pandas DataFrames to concatenate.

Returns A pandas DataFrame

Return type pandas.DataFrame

classmethod finalize(*partitions*)
Perform all deferred calls on partitions.

Parameters **partitions** (*np.ndarray*) – Partitions of Modin Dataframe on which all deferred calls should be performed.

classmethod from_arrow(*at*, *return_dims=False*)
Return the partitions from Apache Arrow (PyArrow).

Parameters

- **at** (*pyarrow.table*) – Arrow Table.
- **return_dims** (*bool, default: False*) – If it's True, return as (np.ndarray, row_lengths, col_widths), else np.ndarray.

Returns A NumPy array with partitions (with dimensions or not).

Return type np.ndarray or (np.ndarray, row_lengths, col_widths)

classmethod from_pandas(*df*, *return_dims=False*)
Return the partitions from pandas.DataFrame.

Parameters

- **df** (*pandas.DataFrame*) – A pandas.DataFrame.
- **return_dims** (*bool, default: False*) – If it's True, return as (np.ndarray, row_lengths, col_widths), else np.ndarray.

Returns A NumPy array with partitions (with dimensions or not).

Return type np.ndarray or (np.ndarray, row_lengths, col_widths)

classmethod get_indices(*axis*, *partitions*, *index_func=None*)
Get the internal indices stored in the partitions.

Parameters

- **axis** (*{0, 1}*) – Axis to extract the labels over.
- **partitions** (*np.ndarray*) – NumPy array with PandasDataframePartition's.
- **index_func** (*callable, default: None*) – The function to be used to extract the indices.

Returns A pandas Index object.

Return type pandas.Index

Notes

These are the global indices of the object. This is mostly useful when you have deleted rows/columns internally, but do not know which ones were deleted.

classmethod groupby_reduce(*axis*, *partitions*, *by*, *map_func*, *reduce_func*, *apply_indices=None*)
Groupby data using the *map_func* provided along the *axis* over the *partitions* then reduce using *reduce_func*.

Parameters

- **axis** (*{0, 1}*) – Axis to groupby over.
- **partitions** (*NumPy 2D array*) – Partitions of the ModinFrame to groupby.
- **by** (*NumPy 2D array*) – Partitions of 'by' to broadcast.
- **map_func** (*callable*) – Map function.
- **reduce_func** (*callable,*) – Reduce function.
- **apply_indices** (*list of ints, default: None*) – Indices of *axis ^ 1* to apply function over.

Returns Partitions with applied groupby.

Return type NumPy array

classmethod lazy_map_partitions(*partitions*, *map_func*)
Apply *map_func* to every partition in *partitions lazily*.

Parameters

- **partitions** (*NumPy 2D array*) – Partitions of Modin Frame.
- **map_func** (*callable*) – Function to apply.

Returns An array of partitions

Return type NumPy array

classmethod map_axis_partitions(*axis*, *partitions*, *map_func*, *keep_partitioning=False*, *lengths=None*, *enumerate_partitions=False*, ***kwargs*)
Apply *map_func* to every partition in *partitions* along given *axis*.

Parameters

- **axis** (*{0, 1}*) – Axis to perform the map across (0 - index, 1 - columns).
- **partitions** (*NumPy 2D array*) – Partitions of Modin Frame.
- **map_func** (*callable*) – Function to apply.
- **keep_partitioning** (*bool, default: False*) – Whether to keep partitioning for Modin Frame. Setting it to True stops data shuffling between partitions.
- **lengths** (*list of ints, default: None*) – List of lengths to shuffle the object.
- **enumerate_partitions** (*bool, default: False*) – Whether or not to pass partition index into *map_func*. Note that *map_func* must be able to accept *partition_idx* kwarg.

- ****kwargs** (*dict*) – Additional options that could be used by different engines.

Returns An array of new partitions for Modin Frame.

Return type NumPy array

Notes

This method should be used in the case when *map_func* relies on some global information about the axis.

classmethod map_partitions(*partitions*, *map_func*)
Apply *map_func* to every partition in *partitions*.

Parameters

- **partitions** (*NumPy 2D array*) – Partitions housing the data of Modin Frame.
- **map_func** (*callable*) – Function to apply.

Returns An array of partitions

Return type NumPy array

classmethod preprocess_func(*map_func*)
Preprocess a function to be applied to *PandasDataframePartition* objects.

Parameters **map_func** (*callable*) – The function to be preprocessed.

Returns The preprocessed version of the *map_func* provided.

Return type callable

Notes

Preprocessing does not require any specific format, only that the *PandasDataframePartition.apply* method will recognize it (for the subclass being used).

If your *PandasDataframePartition* objects assume that a function provided is serialized or wrapped or in some other format, this is the place to add that logic. It is possible that this can also just return *map_func* if the *apply* method of the *PandasDataframePartition* object you are using does not require any modification to a given function.

classmethod row_partitions(*partitions*)
List of *BaseDataframeAxisPartition* objects representing row-wise partitions.

Parameters **partitions** (*list-like*) – List of (smaller) partitions to be combined to row-wise partitions.

Returns A list of *BaseDataframeAxisPartition* objects.

Return type list

Notes

Each value in this list will an *BaseDataframeAxisPartition* object. *BaseDataframeAxisPartition* is located in *axis_partition.py*.

classmethod to_numpy(*partitions*, ***kwargs*)
Convert NumPy array of PandasDataframePartition to NumPy array of data stored within *partitions*.

Parameters

- **partitions** (*np.ndarray*) – NumPy array of PandasDataframePartition.
- ****kwargs** (*dict*) – Keyword arguments for PandasDataframePartition.to_numpy function.

Returns A NumPy array.

Return type np.ndarray

classmethod to_pandas(*partitions*)
Convert NumPy array of PandasDataframePartition to pandas DataFrame.

Parameters **partitions** (*np.ndarray*) – NumPy array of PandasDataframePartition.

Returns A pandas DataFrame

Return type pandas.DataFrame

Generic Ray-based members

Objects which are storage format agnostic but require specific Ray implementation are placed in `modin.core.execution.ray.generic`.

Their purpose is to implement certain parallel I/O operations and to serve as a foundation for building storage format specific objects:

- *RayIO* – implements parallel *to_csv()* and *to_sql()*.
- *GenericRayDataframePartitionManager* – implements parallel *to_numpy()*.

class modin.core.execution.ray.generic.io.io.**RayIO**
Base class for doing I/O operations over Ray.

classmethod to_csv(*qc*, ***kwargs*)
Write records stored in the *qc* to a CSV file.

Parameters

- **qc** (BaseQueryCompiler) – The query compiler of the Modin dataframe that we want to run `to_csv` on.
- ****kwargs** (*dict*) – Parameters for `pandas.to_csv(**kwargs)`.

classmethod to_parquet(*qc*, ***kwargs*)
Write a `DataFrame` to the binary parquet format.

Parameters

- **qc** (BaseQueryCompiler) – The query compiler of the Modin dataframe that we want to run *to_parquet* on.
- ****kwargs** (*dict*) – Parameters for *pandas.to_parquet(**kwargs)*.

classmethod to_sql(*qc*, ***kwargs*)
Write records stored in the *qc* to a SQL database.

Parameters

- **qc** (BaseQueryCompiler) – The query compiler of the Modin dataframe that we want to run to_sql on.
- ****kwargs** (*dict*) – Parameters for pandas.to_sql(**kwargs).

class modin.core.execution.ray.generic.partitioning.partition_manager.**GenericRayDataframePartitionManager**

The class implements the interface in *PandasDataframePartitionManager*.

classmethod to_numpy(*partitions*, ***kwargs*)

Convert *partitions* into a NumPy array.

Parameters

- **partitions** (*NumPy array*) – A 2-D array of partitions to convert to local NumPy array.
- ****kwargs** (*dict*) – Keyword arguments to pass to each partition .to_numpy() call.

Returns

Return type NumPy array

PandasOnRay Dataframe implementation

Modin implements Dataframe, PartitionManager, AxisPartition and Partition classes specific for PandasOnRay execution:

- *PandasOnRayDataframe*
- *PandasOnRayDataframePartition*
- *PandasOnRayDataframeAxisPartition*
- *PandasOnRayDataframePartitionManager*

PandasOnRayDataframe

The class is specific implementation of *PandasDataframe* class using Ray distributed engine. It serves as an intermediate level between *PandasQueryCompiler* and *PandasOnRayDataframePartitionManager*.

Public API

class modin.core.execution.ray.implementations.pandas_on_ray.dataframe.dataframe.**PandasOnRayDataframe**(*pa*
in
de
co
ro
co
un
di

The class implements the interface in PandasDataframe using Ray.

Parameters

- **partitions** (*np.ndarray*) – A 2D NumPy array of partitions.

- **index** (*sequence*) – The index for the dataframe. Converted to a `pandas.Index`.
- **columns** (*sequence*) – The columns object for the dataframe. Converted to a `pandas.Index`.
- **row_lengths** (*list, optional*) – The length of each partition in the rows. The "height" of each of the block partitions. Is computed if not provided.
- **column_widths** (*list, optional*) – The width of each partition in the columns. The "width" of each of the block partitions. Is computed if not provided.
- **dtypes** (*pandas.Series, optional*) – The data types for the dataframe columns.

classmethod combine_dtypes(*list_of_dtypes*, *column_names*)

Describe how data types should be combined when they do not match.

Parameters

- **list_of_dtypes** (*list*) – A list of `pandas.Series` with the data types.
- **column_names** (*list*) – The names of the columns that the data types map to.

Returns A `pandas.Series` containing the finalized data types.

Return type pandas.Series

PandasOnRayDataframePartition

The class is the specific implementation of *PandasDataframePartition*, providing the API to perform operations on a block partition, namely, `pandas.DataFrame`, using Ray as an execution engine.

In addition to wrapping a `pandas.DataFrame`, the class also holds the following metadata:

- `length` - length of `pandas.DataFrame` wrapped
- `width` - width of `pandas.DataFrame` wrapped
- `ip` - node IP address that holds `pandas.DataFrame` wrapped

An operation on a block partition can be performed in two modes:

- asynchronously - via `apply()`
- lazily - via `add_to_apply_calls()`

Public API

class modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.**PandasOnRayDataframe**

The class implements the interface in `PandasDataframePartition`.

Parameters

- **object_id** (*ray.ObjectRef*) – A reference to `pandas.DataFrame` that need to be wrapped with this class.
- **length** (*ray.ObjectRef or int, optional*) – Length or reference to it of wrapped `pandas.DataFrame`.

- **width** (*ray.ObjectRef or int, optional*) – Width or reference to it of wrapped pandas.DataFrame.
- **ip** (*ray.ObjectRef or str, optional*) – Node IP address or reference to it that holds wrapped pandas.DataFrame.
- **call_queue** (*list*) – Call queue that needs to be executed on wrapped pandas.DataFrame.

add_to_apply_calls(*func*, **args*, ***kwargs*)
Add a function to the call queue.

Parameters

- **func** (*callable or ray.ObjectRef*) – Function to be added to the call queue.
- ***args** (*iterable*) – Additional positional arguments to be passed in *func*.
- ****kwargs** (*dict*) – Additional keyword arguments to be passed in *func*.

Returns A new PandasOnRayDataframePartition object.

Return type *PandasOnRayDataframePartition*

Notes

It does not matter if *func* is callable or an ray.ObjectRef. Ray will handle it correctly either way. The keyword arguments are sent as a dictionary.

apply(*func*, **args*, ***kwargs*)
Apply a function to the object wrapped by this partition.

Parameters

- **func** (*callable or ray.ObjectRef*) – A function to apply.
- ***args** (*iterable*) – Additional positional arguments to be passed in *func*.
- ****kwargs** (*dict*) – Additional keyword arguments to be passed in *func*.

Returns A new PandasOnRayDataframePartition object.

Return type *PandasOnRayDataframePartition*

Notes

It does not matter if *func* is callable or an ray.ObjectRef. Ray will handle it correctly either way. The keyword arguments are sent as a dictionary.

drain_call_queue()
Execute all operations stored in the call queue on the object wrapped by this partition.

classmethod empty()
Create a new partition that wraps an empty pandas DataFrame.

Returns A new PandasOnRayDataframePartition object.

Return type *PandasOnRayDataframePartition*

get()
Get the object wrapped by this partition out of the Plasma store.

Returns The object from the Plasma store.

Return type pandas.DataFrame

ip()
Get the node IP address of the object wrapped by this partition.

Returns IP address of the node that holds the data.

Return type str

length()
Get the length of the object wrapped by this partition.

Returns The length of the object.

Return type int

mask(*row_indices*, *col_indices*)
Lazily create a mask that extracts the indices provided.

Parameters

- **row_indices** (*list-like, slice or label*) – The indices for the rows to extract.
- **col_indices** (*list-like, slice or label*) – The indices for the columns to extract.

Returns A new PandasOnRayDataframePartition object.

Return type *PandasOnRayDataframePartition*

classmethod preprocess_func(*func*)
Put a function into the Plasma store to use in apply.

Parameters **func** (*callable*) – A function to preprocess.

Returns A reference to *func*.

Return type ray.ObjectRef

classmethod put(*obj*)
Put an object into Plasma store and wrap it with partition object.

Parameters **obj** (*any*) – An object to be put.

Returns A new PandasOnRayDataframePartition object.

Return type *PandasOnRayDataframePartition*

to_numpy(***kwargs*)
Convert the object wrapped by this partition to a NumPy array.

Parameters ****kwargs** (*dict*) – Additional keyword arguments to be passed in to_numpy.

Returns

Return type np.ndarray

to_pandas()
Convert the object wrapped by this partition to a pandas.DataFrame.

Returns

Return type pandas DataFrame.

wait()
Wait completing computations on the object wrapped by the partition.

width()
Get the width of the object wrapped by the partition.

Returns The width of the object.

Return type int

PandasOnRayDataframeAxisPartition

This class is the specific implementation of *PandasDataframeAxisPartition*, providing the API to perform operations on an axis partition, using Ray as an execution engine. The axis partition is a wrapper over a list of block partitions that are stored in this class.

Public API

class modin.core.execution.ray.implementations.pandas_on_ray.partitioning.axis_partition.**PandasOnRayData**

The class implements the interface in PandasDataframeAxisPartition.

Parameters

- **list_of_blocks** (*list*) – List of PandasOnRayDataframePartition objects.
- **get_ip** (*bool, default: False*) – Whether to get node IP addresses to conforming partitions or not.

classmethod deploy_axis_func(*axis*, *func*, *num_splits*, *kwargs*, *maintain_partitioning*, **partitions*)
Deploy a function along a full axis.

Parameters

- **axis** (*{0, 1}*) – The axis to perform the function along.
- **func** (*callable*) – The function to perform.
- **num_splits** (*int*) – The number of splits to return (see split_result_of_axis_func_pandas).
- **kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.
- **maintain_partitioning** (*bool*) – If True, keep the old partitioning if possible. If False, create a new partition layout.
- ***partitions** (*iterable*) – All partitions that make up the full axis (row or column).

Returns A list of pandas.DataFrame-s.

Return type list

classmethod deploy_func_between_two_axis_partitions(*axis*, *func*, *num_splits*, *len_of_left*, *other_shape*, *kwargs*, **partitions*)
Deploy a function along a full axis between two data sets.

Parameters

- **axis** (*{0, 1}*) – The axis to perform the function along.
- **func** (*callable*) – The function to perform.
- **num_splits** (*int*) – The number of splits to return (see split_result_of_axis_func_pandas).

- **len_of_left** (*int*) – The number of values in *partitions* that belong to the left data set.
- **other_shape** (*np.ndarray*) – The shape of right frame in terms of partitions, i.e. (other_shape[i-1], other_shape[i]) will indicate slice to restore i-1 axis partition.
- **kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.
- ***partitions** (*iterable*) – All partitions that make up the full axis (row or column) for both data sets.

Returns A list of `pandas.DataFrame`-s.

Return type list

instance_type
alias of `ray._raylet.ObjectRef`

partition_type
alias of *modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.PandasOnRayDataframePartition*

PandasOnRayDataframeColumnPartition

Public API

class modin.core.execution.ray.implementations.pandas_on_ray.partitioning.axis_partition.**PandasOnRayData**

The column partition implementation.

All of the implementation for this class is in the parent class, and this class defines the axis to perform the computation over.

Parameters

- **list_of_blocks** (*list*) – List of `PandasOnRayDataframePartition` objects.
- **get_ip** (*bool, default: False*) – Whether to get node IP addresses to conforming partitions or not.

PandasOnRayDataframeRowPartition

Public API

class modin.core.execution.ray.implementations.pandas_on_ray.partitioning.axis_partition.**PandasOnRayData**

The row partition implementation.

All of the implementation for this class is in the parent class, and this class defines the axis to perform the computation over.

Parameters

- **list_of_blocks** (*list*) – List of `PandasOnRayDataframePartition` objects.
- **get_ip** (*bool, default: False*) – Whether to get node IP addresses to conforming partitions or not.

PandasOnRayDataframePartitionManager

This class is the specific implementation of *GenericRayDataframePartitionManager* using Ray distributed engine. This class is responsible for partition manipulation and applying a funcion to block/row/column partitions.

Public API

class modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition_manager.**PandasOnRayDataframePartitionManager**

The class implements the interface in *PandasDataframePartitionManager*.

classmethod apply_func_to_indices_both_axis(*partitions*, *func*, *row_partitions_list*, *col_partitions_list*, *item_to_distribute=None*, *row_lengths=None*, *col_widths=None*)

Apply a function along both axes.

Parameters

- **partitions** (*np.ndarray*) – The partitions to which the *func* will apply.
- **func** (*callable*) – The function to apply.
- **row_partitions_list** (*list*) – List of row partitions.
- **col_partitions_list** (*list*) – List of column partitions.
- **item_to_distribute** (*item, optional*) – The item to split up so it can be applied over both axes.
- **row_lengths** (*list of ints, optional*) – Lengths of partitions for every row. If not specified this information is extracted from partitions itself.
- **col_widths** (*list of ints, optional*) – Widths of partitions for every column. If not specified this information is extracted from partitions itself.

Returns A NumPy array with partitions.

Return type np.ndarray

Notes

For your func to operate directly on the indices provided, it must use `row_internal_indices` and `col_internal_indices` as keyword arguments.

classmethod apply_func_to_select_indices(*axis*, *partitions*, *func*, *indices*, *keep_remaining=False*)

Apply a *func* to select *indices* of *partitions*.

Parameters

- **axis** (*{0, 1}*) – Axis to apply the *func* over.
- **partitions** (*np.ndarray*) – The partitions to which the *func* will apply.
- **func** (*callable*) – The function to apply to these indices of partitions.
- **indices** (*dict*) – The indices to apply the function to.
- **keep_remaining** (*bool, default: False*) – Whether or not to keep the other partitions. Some operations may want to drop the remaining partitions and keep only the results.

Returns A NumPy array with partitions.

Return type np.ndarray

Notes

Your internal function must take a kwarg *internal_indices* for this to work correctly. This prevents information leakage of the internal index to the external representation.

classmethod apply_func_to_select_indices_along_full_axis(*axis*, *partitions*, *func*, *indices*, *keep_remaining=False*)

Apply a *func* to a select subset of full columns/rows.

Parameters

- **axis** (*{0, 1}*) – The axis to apply the *func* over.
- **partitions** (*np.ndarray*) – The partitions to which the *func* will apply.
- **func** (*callable*) – The function to apply.
- **indices** (*list-like*) – The global indices to apply the func to.
- **keep_remaining** (*bool, default: False*) – Whether or not to keep the other partitions. Some operations may want to drop the remaining partitions and keep only the results.

Returns A NumPy array with partitions.

Return type np.ndarray

Notes

This should be used when you need to apply a function that relies on some global information for the entire column/row, but only need to apply a function to a subset. For your func to operate directly on the indices provided, it must use *internal_indices* as a keyword argument.

classmethod binary_operation(*axis*, *left*, *func*, *right*)

Apply a function that requires partitions of two `PandasOnRayDataframe` objects.

Parameters

- **axis** (*{0, 1}*) – The axis to apply the function over (0 - rows, 1 - columns).
- **left** (*np.ndarray*) – The partitions of left `PandasOnRayDataframe`.
- **func** (*callable*) – The function to apply.
- **right** (*np.ndarray*) – The partitions of right `PandasOnRayDataframe`.

Returns A NumPy array with new partitions.

Return type np.ndarray

classmethod broadcast_apply(*axis*, *apply_func*, *left*, *right*, *other_name='r'*)

Broadcast the *right* partitions to *left* and apply *apply_func* to selected indices.

Parameters

- **axis** (*{0, 1}*) – Axis to apply and broadcast over.
- **apply_func** (*callable*) – Function to apply.
- **left** (*np.ndarray*) – NumPy 2D array of left partitions.

- **right** (*np.ndarray*) – NumPy 2D array of right partitions.
- **other_name** (*str, default: "r"*) – Name of key-value argument for *apply_func* that is used to pass *right* to *apply_func*.

Returns An array of partition objects.

Return type np.ndarray

classmethod get_indices(*axis*, *partitions*, *index_func=None*)

Get the internal indices stored in the partitions.

Parameters

- **axis** (*{0, 1}*) – Axis to extract the labels over.
- **partitions** (*np.ndarray*) – NumPy array with PandasDataframePartition-s.
- **index_func** (*callable, default: None*) – The function to be used to extract the indices.

Returns A pandas.Index object.

Return type pandas.Index

Notes

These are the global indices of the object. This is mostly useful when you have deleted rows/columns internally, but do not know which ones were deleted.

classmethod lazy_map_partitions(*partitions*, *map_func*)

Apply *map_func* to every partition in *partitions lazily*.

Parameters

- **partitions** (*np.ndarray*) – A NumPy 2D array of partitions to perform operation on.
- **map_func** (*callable*) – Function to apply.

Returns A NumPy array of partitions.

Return type np.ndarray

classmethod map_axis_partitions(*axis*, *partitions*, *map_func*, *keep_partitioning=False*, *lengths=None*, *enumerate_partitions=False*, ***kwargs*)

Apply *map_func* to every partition in *partitions* along given *axis*.

Parameters

- **axis** (*{0, 1}*) – Axis to perform the map across (0 - index, 1 - columns).
- **partitions** (*np.ndarray*) – A NumPy 2D array of partitions to perform operation on.
- **map_func** (*callable*) – Function to apply.
- **keep_partitioning** (*bool, default: False*) – Whether to keep partitioning for Modin Frame. Setting it to True prevents data shuffling between partitions.
- **lengths** (*list of ints, default: None*) – List of lengths to shuffle the object.
- **enumerate_partitions** (*bool, default: False*) – Whether or not to pass partition index into *map_func*. Note that *map_func* must be able to accept *partition_idx* kwarg.

- ****kwargs** (*dict*) – Additional options that could be used by different engines.

Returns A NumPy array of new partitions for Modin Frame.

Return type np.ndarray

Notes

This method should be used in the case when *map_func* relies on some global information about the axis.

classmethod map_partitions(*partitions*, *map_func*)
Apply *map_func* to every partition in *partitions*.

Parameters

- **partitions** (*np.ndarray*) – A NumPy 2D array of partitions to perform operation on.
- **map_func** (*callable*) – Function to apply.

Returns A NumPy array of partitions.

Return type np.ndarray

cuDFOnRay Dataframe Implementation

Modin implements `Dataframe`, `PartitionManager`, `AxisPartition`, `Partition` and `GPUManager` classes specific for `cuDFOnRay` execution:

- *cuDFOnRayDataframe*
- *cuDFOnRayDataframePartition*
- *cuDFOnRayDataframeAxisPartition*
- *cuDFOnRayDataframePartitionManager*
- *GPUManager*

cuDFOnRayDataframe

The class is the specific implementation of *PandasDataframe* class using Ray distributed engine. It serves as an intermediate level between `cuDFQueryCompiler` and *cuDFOnRayDataframePartitionManager*.

Public API

class modin.core.execution.ray.implementations.cudf_on_ray.dataframe.dataframe.**cuDFOnRayDataframe**(*partitio*
in-
dex,
column
row_le
col-
umn_w
dtypes=

The class implements the interface in `PandasOnRayDataframe` using cuDF.

Parameters

- **partitions** (*np.ndarray*) – A 2D NumPy array of partitions.
- **index** (*sequence*) – The index for the dataframe. Converted to a `pandas.Index`.
- **columns** (*sequence*) – The columns object for the dataframe. Converted to a `pandas.Index`.
- **row_lengths** (*list, optional*) – The length of each partition in the rows. The "height" of each of the block partitions. Is computed if not provided.
- **column_widths** (*list, optional*) – The width of each partition in the columns. The "width" of each of the block partitions. Is computed if not provided.
- **dtypes** (*pandas.Series, optional*) – The data types for the dataframe columns.

mask(*row_indices=None*, *row_numeric_idx=None*, *col_indices=None*, *col_numeric_idx=None*)
Lazily select columns or rows from given indices.

Parameters

- **row_indices** (*list of hashable, optional*) – The row labels to extract.
- **row_numeric_idx** (*list of int, optional*) – The row indices to extract.
- **col_indices** (*list of hashable, optional*) – The column labels to extract.
- **col_numeric_idx** (*list of int, optional*) – The column indices to extract.

Returns A new `cuDFOnRayDataframe` from the mask provided.

Return type *cuDFOnRayDataframe*

Notes

If both *row_indices* and *row_numeric_idx* are set, *row_indices* will be used. The same rule applied to *col_indices* and *col_numeric_idx*.

synchronize_labels(*axis=None*)
Synchronize labels by applying the index object (Index or Columns) to the partitions eagerly.

Parameters **axis** (*{0, 1, None}, default: None*) – The axis to apply to. If None, it applies to both axes.

cuDFOnRayDataframePartition

The class is the specific implementation of *PandasDataframePartition*, providing the API to perform operations on a block partition, namely, `cudf.DataFrame`, using Ray as an execution engine.

An operation on a block partition can be performed asynchronously in two ways:

- *apply()* returns `ray.ObjectRef` with integer key of operation result from internal storage.
- `add_to_apply_calls()` returns a new *cuDFOnRayDataframePartition* object that is based on result of operation.

Public API

class modin.core.execution.ray.implementations.cudf_on_ray.partitioning.partition.cuDFOnRayDataframePar

The class implements the interface in PandasDataframePartition using cuDF on Ray.

Parameters

- **gpu_manager** (*modin.core.execution.ray.implementations.cudf_on_ray.partitioning.GPUManager*) – A gpu manager to store cuDF dataframes.
- **key** (*ray.ObjectRef or int*) – An integer key (or reference to key) associated with cudf.DataFrame stored in *gpu_manager*.
- **length** (*ray.ObjectRef or int, optional*) – Length or reference to it of wrapped pandas.DataFrame.
- **width** (*ray.ObjectRef or int, optional*) – Width or reference to it of wrapped pandas.DataFrame.

add_to_apply_calls(*func*, ***kwargs*)

Apply *func* to this partition and create new.

Parameters

- **func** (*callable*) – A function to apply.
- ****kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.

Returns New partition based on result of *func*.

Return type *cuDFOnRayDataframePartition*

Notes

We eagerly schedule the apply *func* and produce a new cuDFOnRayDataframePartition.

apply(*func*, ***kwargs*)

Apply *func* to this partition.

Parameters

- **func** (*callable*) – A function to apply.
- ****kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.

Returns A reference to integer key of result in internal dict-storage of *self.gpu_manager*.

Return type ray.ObjectRef

apply_result_not_dataframe(*func*, ***kwargs*)

Apply *func* to this partition.

Parameters

- **func** (*callable*) – A function to apply.
- ****kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.

Returns A reference to integer key of result in internal dict-storage of *self.gpu_manager*.

Return type ray.ObjectRef

copy()
Create a full copy of this object.

Returns

Return type *cuDFOnRayDataframePartition*

free()
Free the dataFrame and associated *self.key* out of *self.gpu_manager*.

get()
Get object stored by this partition from *self.gpu_manager*.

Returns

Return type ray.ObjectRef

get_gpu_manager()
Get gpu manager associated with this partition.

Returns GPUManager associated with this object.

Return type modin.core.execution.ray.implementations.cudf_on_ray.partitioning.GPUManager

get_key()
Get integer key of this partition in dict-storage of *self.gpu_manager*.

Returns

Return type int

get_object_id()
Get object stored for this partition from *self.gpu_manager*.

Returns

Return type ray.ObjectRef

length()
Get the length of the object wrapped by this partition.

Returns The length (or reference to length) of the object.

Return type int or ray.ObjectRef

mask(*row_indices*, *col_indices*)
Select columns or rows from given indices.

Parameters

- **row_indices** (*list of hashable*) – The row labels to extract.
- **col_indices** (*list of hashable*) – The column labels to extract.

Returns A reference to integer key of result in internal dict-storage of *self.gpu_manager*.

Return type ray.ObjectRef

classmethod preprocess_func(*func*)
Put *func* to Ray object store.

Parameters **func** (*callable*) – Function to put.

Returns A reference to *func* in Ray object store.

Return type ray.ObjectRef

classmethod put(*gpu_manager*, *pandas_dataframe*)
Put *pandas_dataframe* to *gpu_manager*.

Parameters

- **gpu_manager** (*modin.core.execution.ray.implementations.cudf_on_ray.partitioning.GPUManager*) – A gpu manager to store cuDF dataframes.
- **pandas_dataframe** (*pandas.DataFrame/pandas.Series*) – A pandas.DataFrame/pandas.Series to put.

Returns A reference to integer key of added pandas.DataFrame to internal dict-storage in *gpu_manager*.

Return type ray.ObjectRef

to_numpy()
Convert this partition to NumPy array.

Returns

Return type NumPy array

to_pandas()
Convert this partition to pandas.DataFrame.

Returns

Return type pandas.DataFrame

width()
Get the width of the object wrapped by this partition.

Returns The width (or reference to width) of the object.

Return type int or ray.ObjectRef

cuDFOnRayDataframeAxisPartition

The class is a base class for any axis partition class based on Ray engine and cuDF storage format. This class provides the API to perform operations on an axis partition, using Ray as the execution engine. The axis partition is made up of list of block partitions that are stored in this class.

Public API

class modin.core.execution.ray.implementations.cudf_on_ray.partitioning.axis_partition.**cuDFOnRayDatafra**
Base class for any axis partition class for cuDF storage format.

Parameters **partitions** (*np.ndarray*) – NumPy array with cuDFOnRayDataframePartition-s.

partition_type
alias of *modin.core.execution.ray.implementations.cudf_on_ray.partitioning.partition.cuDFOnRayDataframePartition*

cuOnRayDataframeColumnPartition

Public API

class modin.core.execution.ray.implementations.cudf_on_ray.partitioning.axis_partition.**cuDFOnRayDataframeColumnPartition**
The column partition implementation of cuDFOnRayDataframeAxisPartition.

Parameters **partitions** (*np.ndarray*) – NumPy array with cuDFOnRayDataframePartition-s.

reduce(*func*)
Reduce partitions along *self.axis* and apply *func*.

Parameters **func** (*callable*) – A func to apply.

Returns

Return type *cuDFOnRayDataframePartition*

cuDFOnRayDataframeRowPartition

Public API

class modin.core.execution.ray.implementations.cudf_on_ray.partitioning.axis_partition.**cuDFOnRayDataframeRowPartition**
The row partition implementation of cuDFOnRayDataframeAxisPartition.

Parameters **partitions** (*np.ndarray*) – NumPy array with cuDFOnRayDataframePartition-s.

reduce(*func*)
Reduce partitions along *self.axis* and apply *func*.

Parameters **func** (*calalble*) – A func to apply.

Returns

Return type *cuDFOnRayDataframePartition*

Notes

Since we are using row partitions, we can bypass the Ray plasma store during axis reduction functions.

cuDFOnRayDataframePartitionManager

This class is the specific implementation of *GenericRayDataframePartitionManager*. It serves as an intermediate level between *cuDFOnRayDataframe* and *cuDFOnRayDataframePartition* class. This class is responsible for partition manipulation and applying a function to block/row/column partitions.

Public API

class modin.core.execution.ray.implementations.cudf_on_ray.partitioning.partition_manager.**cuDFOnRayDataframePartitionManager**

The class implements the interface in `GenericRayDataframePartitionManager` using cuDF on Ray.

classmethod from_pandas(*df*, *return_dims=False*)

Create partitions from `pandas.DataFrame/pandas.Series`.

Parameters

- **df** (*pandas.DataFrame/pandas.Series*) – A `pandas.DataFrame` to add.
- **return_dims** (*boolean, default: False*) – Is return dimensions or not.

Returns List of partitions in case *return_dims* == False, tuple (partitions, row lengths, col widths) in other case.

Return type list or tuple

classmethod lazy_map_partitions(*partitions*, *map_func*)

Apply *map_func* to every partition lazily.

Compared to Modin-CPU, Modin-GPU lazy version represents:

(1) A scheduled function in the Ray task graph.

(2) A non-materialized key.

Parameters

- **partitions** (*np.ndarray*) – NumPy array with partitions.
- **map_func** (*callable*) – The function to apply.

Returns A NumPy array of `cuDFOnRayDataframePartition` objects.

Return type np.ndarray

GPUManager

The Ray actor-class stores `cuDF.DataFrame`-s and executes operations on it.

Public API

class modin.core.execution.ray.implementations.cudf_on_ray.partitioning.gpu_manager.**GPUManager**(*gpu_id*)

Ray actor-class to store `cudf.DataFrame`-s and execute functions on it.

Parameters **gpu_id** (*int*) – The identifier of GPU.

apply(*first*, *other*, *func*, ***kwargs*)

Apply *func* to values associated with *first*/*other* keys of *self.cudf_dataframe_dict* with storing of the result.

Store the return value of *func* (a new `cudf.DataFrame`) into *self.cudf_dataframe_dict*.

Parameters

- **first** (*int*) – The first key associated with dataframe from *self.cudf_dataframe_dict*.
- **other** (*int or ray.ObjectRef*) – The second key associated with dataframe from *self.cudf_dataframe_dict*. If it isn't a real key, the *func* will be applied to the *first* only.

- **func** (*callable*) – A function to apply.
- ****kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.

Returns The new key of the new dataframe stored in *self.cudf_dataframe_dict* (will be a `ray.ObjectRef` in outside level).

Return type int

apply_non_persistent(*first*, *other*, *func*, ***kwargs*)

Apply *func* to values associated with *first*/*other* keys of *self.cudf_dataframe_dict*.

Parameters

- **first** (*int*) – The first key associated with dataframe from *self.cudf_dataframe_dict*.
- **other** (*int*) – The second key associated with dataframe from *self.cudf_dataframe_dict*. If it isn't a real key, the *func* will be applied to the *first* only.
- **func** (*callable*) – A function to apply.
- ****kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.

Returns The result of the *func* (will be a `ray.ObjectRef` in outside level).

Return type The type of return of *func*

free(*key*)

Free the dataFrame and associated *key* out of *self.cudf_dataframe_dict*.

Parameters **key** (*int*) – The key to be deleted.

get_id()

Get the *self.gpu_id* from this object.

Returns The gpu_id from this object (will be a `ray.ObjectRef` in outside level).

Return type int

get_oid(*key*)

Get the value from *self.cudf_dataframe_dict* by *key*.

Parameters **key** (*int*) – The key to get value.

Returns Dataframe corresponding to *key* (*will be a ``ray.ObjectRef`` in outside level).

Return type cudf.DataFrame

put(*pandas_df*)

Convert *pandas_df* to `cudf.DataFrame` and put it to *self.cudf_dataframe_dict*.

Parameters **pandas_df** (*pandas.DataFrame/pandas.Series*) – A pandas DataFrame/Series to be added.

Returns The key associated with added dataframe (will be a `ray.ObjectRef` in outside level).

Return type int

reduce(*first*, *others*, *func*, *axis=0*, ***kwargs*)

Apply *func* to values associated with *first* key and *others* keys of *self.cudf_dataframe_dict* with storing of the result.

Dataframes associated with *others* keys will be concatenated to one dataframe.

Store the return value of *func* (a new `cudf.DataFrame`) into *self.cudf_dataframe_dict*.

Parameters

- **first** (*int*) – The first key associated with dataframe from *self.cudf_dataframe_dict*.
- **others** (*list of int / list of ray.ObjectRef*) – The list of keys associated with dataframe from *self.cudf_dataframe_dict*.
- **func** (*callable*) – A function to apply.
- **axis** (*{0, 1}, default: 0*) – An axis corresponding to a particular row/column of the dataframe.
- ****kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.

Returns The new key of the new dataframe stored in *self.cudf_dataframe_dict* (will be a `ray.ObjectRef` in outside level).

Return type int

Notes

If `len(others) == 0` *func* should be able to work with 2nd positional argument with None value.

store_new_df(*df*)

Store *df* in *self.cudf_dataframe_dict*.

Parameters **df** (*cudf.DataFrame*) – The `cudf.DataFrame` to be added.

Returns The key associated with added dataframe (will be a `ray.ObjectRef` in outside level).

Return type int

PandasOnDask Dataframe implementation

This page describes the implementation of *Modin PandasDataframe Objects* specific for *PandasOnDask* execution.

- *PandasOnDaskDataframe*
- *PandasOnDaskDataframePartition*
- *PandasOnDaskDataframeAxisPartition*
- *PandasOnDaskDataframePartitionManager*

PandasOnDaskDataframe

The class is the specific implementation of the dataframe algebra for the *Dask* execution engine. It serves as an intermediate level between `pandas` query compiler and `PandasOnDaskDataframePartitionManager`.

Public API

class modin.core.execution.dask.implementations.pandas_on_dask.dataframe.dataframe.**PandasOnDaskDataframe**

The class implements the interface in `PandasDataframe`.

Parameters

- **partitions** (*np.ndarray*) – A 2D NumPy array of partitions.
- **index** (*sequence*) – The index for the dataframe. Converted to a pandas.Index.
- **columns** (*sequence*) – The columns object for the dataframe. Converted to a pandas.Index.
- **row_lengths** (*list, optional*) – The length of each partition in the rows. The "height" of each of the block partitions. Is computed if not provided.
- **column_widths** (*list, optional*) – The width of each partition in the columns. The "width" of each of the block partitions. Is computed if not provided.
- **dtypes** (*pandas.Series, optional*) – The data types for the dataframe columns.

PandasOnDaskDataframePartition

The class is the specific implementation of *PandasDataframePartition*, providing the API to perform operations on a block partition, namely, `pandas.DataFrame`, using Dask as the execution engine.

In addition to wrapping a `pandas.DataFrame`, the class also holds the following metadata:

- `length` - length of `pandas.DataFrame` wrapped
- `width` - width of `pandas.DataFrame` wrapped
- `ip` - node IP address that holds `pandas.DataFrame` wrapped

An operation on a block partition can be performed in two modes:

- asynchronously - via `apply()`
- lazily - via `add_to_apply_calls()`

Public API

class modin.core.execution.dask.implementations.pandas_on_dask.partitioning.partition.**PandasOnDaskDataf**

The class implements the interface in `PandasDataframePartition`.

Parameters

- **future** (*distributed.Future*) – A reference to pandas DataFrame that need to be wrapped with this class.
- **length** (*distributed.Future or int, optional*) – Length or reference to it of wrapped pandas DataFrame.
- **width** (*distributed.Future or int, optional*) – Width or reference to it of wrapped pandas DataFrame.
- **ip** (*distributed.Future or str, optional*) – Node IP address or reference to it that holds wrapped pandas DataFrame.
- **call_queue** (*list, optional*) – Call queue that needs to be executed on wrapped pandas DataFrame.

add_to_apply_calls(*func*, **args*, ***kwargs*)
Add a function to the call queue.

Parameters

- **func** (*callable*) – Function to be added to the call queue.
- ***args** (*iterable*) – Additional positional arguments to be passed in *func*.
- ****kwargs** (*dict*) – Additional keyword arguments to be passed in *func*.

Returns A new PandasOnDaskDataframePartition object.

Return type *PandasOnDaskDataframePartition*

Notes

The keyword arguments are sent as a dictionary.

apply(*func*, **args*, ***kwargs*)
Apply a function to the object wrapped by this partition.

Parameters

- **func** (*callable*) – A function to apply.
- ***args** (*iterable*) – Additional positional arguments to be passed in *func*.
- ****kwargs** (*dict*) – Additional keyword arguments to be passed in *func*.

Returns A new PandasOnDaskDataframePartition object.

Return type *PandasOnDaskDataframePartition*

Notes

The keyword arguments are sent as a dictionary.

drain_call_queue()
Execute all operations stored in the call queue on the object wrapped by this partition.

classmethod empty()
Create a new partition that wraps an empty pandas DataFrame.

Returns A new PandasOnDaskDataframePartition object.

Return type *PandasOnDaskDataframePartition*

get()
Get the object wrapped by this partition out of the distributed memory.

Returns The object from the distributed memory.

Return type pandas.DataFrame

ip()
Get the node IP address of the object wrapped by this partition.

Returns IP address of the node that holds the data.

Return type str

length()
Get the length of the object wrapped by this partition.

Returns The length of the object.

Return type int

mask(*row_indices*, *col_indices*)
Lazily create a mask that extracts the indices provided.

Parameters

- **row_indices** (*list-like, slice or label*) – The indices for the rows to extract.
- **col_indices** (*list-like, slice or label*) – The indices for the columns to extract.

Returns A new PandasOnDaskDataframePartition object.

Return type *PandasOnDaskDataframePartition*

classmethod preprocess_func(*func*)
Preprocess a function before an apply call.

Parameters **func** (*callable*) – The function to preprocess.

Returns An object that can be accepted by apply.

Return type callable

classmethod put(*obj*)
Put an object into distributed memory and wrap it with partition object.

Parameters **obj** (*any*) – An object to be put.

Returns A new PandasOnDaskDataframePartition object.

Return type *PandasOnDaskDataframePartition*

to_numpy(***kwargs*)
Convert the object wrapped by this partition to a NumPy array.

Parameters ****kwargs** (*dict*) – Additional keyword arguments to be passed in to_numpy.

Returns

Return type np.ndarray.

to_pandas()
Convert the object wrapped by this partition to a pandas DataFrame.

Returns

Return type pandas.DataFrame

wait()
Wait completing computations on the object wrapped by the partition.

width()
Get the width of the object wrapped by the partition.

Returns The width of the object.

Return type int

PandasOnDaskDataframeAxisPartition

The class is the specific implementation of *PandasDataframeAxisPartition*, providing the API to perform operations on an axis (column or row) partition using Dask as the execution engine. The axis partition is a wrapper over a list of block partitions that are stored in this class.

Public API

class modin.core.execution.dask.implementations.pandas_on_dask.partitioning.axis_partition.**PandasOnDaskD**

The class implements the interface in PandasDataframeAxisPartition.

Parameters

- **list_of_blocks** (*list*) – List of PandasOnDaskDataframePartition objects.
- **get_ip** (*bool, default: False*) – Whether to get node IP addresses of conforming partitions or not.

classmethod deploy_axis_func(*axis*, *func*, *num_splits*, *kwargs*, *maintain_partitioning*, **partitions*)

Deploy a function along a full axis.

Parameters

- **axis** (*{0, 1}*) – The axis to perform the function along.
- **func** (*callable*) – The function to perform.
- **num_splits** (*int*) – The number of splits to return (see *split_result_of_axis_func_pandas*).
- **kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.
- **maintain_partitioning** (*bool*) – If True, keep the old partitioning if possible. If False, create a new partition layout.
- ***partitions** (*iterable*) – All partitions that make up the full axis (row or column).

Returns A list of distributed.Future.

Return type list

classmethod deploy_func_between_two_axis_partitions(*axis*, *func*, *num_splits*, *len_of_left*, *other_shape*, *kwargs*, **partitions*)

Deploy a function along a full axis between two data sets.

Parameters

- **axis** (*{0, 1}*) – The axis to perform the function along.
- **func** (*callable*) – The function to perform.
- **num_splits** (*int*) – The number of splits to return (see *split_result_of_axis_func_pandas*).
- **len_of_left** (*int*) – The number of values in *partitions* that belong to the left data set.
- **other_shape** (*np.ndarray*) – The shape of right frame in terms of partitions, i.e. (other_shape[i-1], other_shape[i]) will indicate slice to restore i-1 axis partition.
- **kwargs** (*dict*) – Additional keywords arguments to be passed in *func*.

- ***partitions** (*iterable*) – All partitions that make up the full axis (row or column) for both data sets.

Returns A list of distributed.Future.

Return type list

instance_type
alias of `distributed.client.Future`

partition_type
alias of *modin.core.execution.dask.implementations.pandas_on_dask.partitioning.partition.PandasOnDaskDataframePartition*

PandasOnDaskDataframeColumnPartition

Public API

class modin.core.execution.dask.implementations.pandas_on_dask.partitioning.axis_partition.**PandasOnDaskD**

The column partition implementation.

All of the implementation for this class is in the parent class, and this class defines the axis to perform the computation over.

Parameters

- **list_of_blocks** (*list*) – List of `PandasOnDaskDataframePartition` objects.
- **get_ip** (*bool, default: False*) – Whether to get node IP addresses to conforming partitions or not.

PandasOnDaskDataframeRowPartition

Public API

class modin.core.execution.dask.implementations.pandas_on_dask.partitioning.axis_partition.**PandasOnDaskD**

The row partition implementation.

All of the implementation for this class is in the parent class, and this class defines the axis to perform the computation over.

Parameters

- **list_of_blocks** (*list*) – List of `PandasOnDaskDataframePartition` objects.
- **get_ip** (*bool, default: False*) – Whether to get node IP addresses to conforming partitions or not.

PandasOnDaskDataframePartitionManager

This class is the specific implementation of *PandasDataframePartitionManager* using Dask as the execution engine. This class is responsible for partition manipulation and applying a funcion to block/row/column partitions.

Public API

class modin.core.execution.dask.implementations.pandas_on_dask.partitioning.partition_manager.**PandasOnDaskDataframePartitionManager**

The class implements the interface in *PandasDataframePartitionManager*.

classmethod broadcast_apply(*axis*, *apply_func*, *left*, *right*, *other_name='r'*)

Broadcast the *right* partitions to *left* and apply *apply_func* function.

Parameters

- **axis** (*{0, 1}*) – Axis to apply and broadcast over.
- **apply_func** (*callable*) – Function to apply.
- **left** (*np.ndarray*) – NumPy array of left partitions.
- **right** (*np.ndarray*) – NumPy array of right partitions.
- **other_name** (*str, default: "r"*) – Name of key-value argument for *apply_func* that is used to pass *right* to *apply_func*.

Returns NumPy array of result partition objects.

Return type np.ndarray

classmethod get_indices(*axis*, *partitions*, *index_func*)

Get the internal indices stored in the partitions.

Parameters

- **axis** (*{0, 1}*) – Axis to extract the labels over.
- **partitions** (*np.ndarray*) – The array of partitions from which need to extract the labels.
- **index_func** (*callable*) – The function to be used to extract the indices.

Returns A pandas Index object.

Return type pandas.Index

Notes

These are the global indices of the object. This is mostly useful when you have deleted rows/columns internally, but do not know which ones were deleted.

Experimental

`modin.experimental` holds experimental functionality that is under development right now and provides a limited set of functionality:

- *xgboost*
- *sklearn*

Scikit-learn module description

This module holds experimental scikit-learn-specific functionality for Modin.

API

Module holds model selection specific functionality.

`modin.experimental.sklearn.model_selection.`**`train_test_split`**(*df*, ***options*)

Split input data to train and test data.

Parameters

- **`df`** (*`modin.pandas.DataFrame / modin.pandas.Series`*) – Data to split.
- **`**options`** (*`dict`*) – Keyword arguments. If *train_size* key isn't provided *train_size* will be 0.75.

Returns A pair of modin.pandas.DataFrame / modin.pandas.Series.

Return type tuple

Modin XGBoost module description

High-level Module Overview

This module holds classes, public interface and internal functions for distributed XGBoost in Modin.

Public classes *`Booster`*, *`DMatrix`* and function *`train()`* provide the user with familiar XGBoost interfaces. They are located in the `modin.experimental.xgboost.xgboost` module.

The internal module `modin.experimental.xgboost.xgboost.xgboost_ray` contains the implementation of Modin XGBoost for the Ray execution engine. This module mainly consists of the Ray actor-class *`ModinXGBoostActor`*, a function to distribute Modin's partitions between actors *`_assign_row_partitions_to_actors()`*, an internal *`_train()`*/*`_predict()`* function used from the public interfaces and additional util functions for computing cluster resources, actor creations etc.

Public interfaces

DMatrix inherits original class `xgboost.DMatrix` and overrides its constructor, which currently supports only *data* and *label* parameters. Both of the parameters must be `modin.pandas.DataFrame`, which will be internally unwrapped to lists of delayed objects of Modin's row partitions using the function *unwrap_partitions()*.

class modin.experimental.xgboost.**DMatrix**(*data*, *label=None*, *missing=None*, *silent=False*, *feature_names=None*, *feature_types=None*, *feature_weights=None*, *enable_categorical=None*)

DMatrix holds references to partitions of Modin DataFrame.

On init stage unwrapping partitions of Modin DataFrame is started.

Parameters

- **data** (*modin.pandas.DataFrame*) – Data source of DMatrix.
- **label** (*modin.pandas.DataFrame or modin.pandas.Series, optional*) – Labels used for training.
- **missing** (*float, optional*) – Value in the input data which needs to be present as a missing value. If `None`, defaults to `np.nan`.
- **silent** (*boolean, optional*) – Whether to print messages during construction or not.
- **feature_names** (*list, optional*) – Set names for features.
- **feature_types** (*list, optional*) – Set types for features.
- **feature_weights** (*array_like, optional*) – Set feature weights for column sampling.
- **enable_categorical** (*boolean, optional*) – Experimental support of specializing for categorical features.

Notes

Currently DMatrix doesn't support *weight*, *base_margin*, *nthread*, *group*, *qid*, *label_lower_bound*, *label_upper_bound* parameters.

property feature_names

Get column labels.

Returns

Return type Column labels.

property feature_types

Get column types.

Returns

Return type Column types.

get_dmatrix_params()

Get dict of DMatrix parameters excluding *self.data*/*self.label*.

Returns

Return type dict

get_float_info(*name*)

Get float property from the DMatrix.

Parameters **name** (*str*) – The field name of the information.

Returns

Return type A NumPy array of float information of the data.

num_col()
Get number of columns.

Returns

Return type int

num_row()
Get number of rows.

Returns

Return type int

set_info(*, *label=None*, *feature_names=None*, *feature_types=None*, *feature_weights=None*) → None
Set meta info for DMatrix.

Parameters

- **label** (*modin.pandas.DataFrame or modin.pandas.Series, optional*) – Labels used for training.
- **feature_names** (*list, optional*) – Set names for features.
- **feature_types** (*list, optional*) – Set types for features.
- **feature_weights** (*array_like, optional*) – Set feature weights for column sampling.

Booster inherits original class `xgboost.Booster` and overrides method `predict`. The difference from original class interface for `predict` method is changing the type of the *data* parameter to *DMatrix*.

class modin.experimental.xgboost.**Booster**(*params=None*, *cache=()*, *model_file=None*)
A Modin Booster of XGBoost.

Booster is the model of XGBoost, that contains low level routines for training, prediction and evaluation.

Parameters

- **params** (*dict, optional*) – Parameters for boosters.
- **cache** (*list, default: empty*) – List of cache items.
- **model_file** (*string/os.PathLike/xgb.Booster/bytearray, optional*) – Path to the model file if it's string or PathLike or xgb.Booster.

predict(*data:* modin.experimental.xgboost.xgboost.DMatrix, ***kwargs*)
Run distributed prediction with a trained booster.

During execution it runs `xgb.predict` on each worker for subset of *data* and creates Modin DataFrame with prediction results.

Parameters

- **data** (modin.experimental.xgboost.DMatrix) – Input data used for prediction.
- ****kwargs** (*dict*) – Other parameters are the same as for `xgboost.Booster.predict`.

Returns Modin DataFrame with prediction results.

Return type modin.pandas.DataFrame

train() function has 2 differences from the original `train` function - (1) the data type of *dtrain* parameter is *DMatrix* and (2) a new parameter *num_actors*.

`modin.experimental.xgboost.train`(*params: Dict*, *dtrain:* modin.experimental.xgboost.xgboost.DMatrix, **args*, *evals=()*, *num_actors: Optional[int] = None*, *evals_result: Optional[Dict] = None*, ***kwargs*)

Run distributed training of XGBoost model.

During work it evenly distributes *dtrain* between workers according to IP addresses partitions (in case of not even distribution of *dtrain* over nodes, some partitions will be re-distributed between nodes), runs xgb.train on each worker for subset of *dtrain* and reduces training results of each worker using Rabit Context.

Parameters

- **params** (*dict*) – Booster params.
- **dtrain** (`modin.experimental.xgboost.DMatrix`) – Data to be trained against.
- ***args** (*iterable*) – Other parameters for *xgboost.train*.
- **evals** (*list of pairs (*`modin.experimental.xgboost.DMatrix`*, str), default: empty*) – List of validation sets for which metrics will evaluated during training. Validation metrics will help us track the performance of the model.
- **num_actors** (*int, optional*) – Number of actors for training. If unspecified, this value will be computed automatically.
- **evals_result** (*dict, optional*) – Dict to store evaluation results in.
- ****kwargs** (*dict*) – Other parameters are the same as *xgboost.train*.

Returns A trained booster.

Return type *modin.experimental.xgboost.Booster*

Internal execution flow on Ray engine

Internal functions *_train()* and *_predict()* work similar to xgboost.

Training

1. The data is passed to *_train()* function as a *DMatrix* object. Using an iterator of *DMatrix*, lists of `ray.ObjectRef` with row partitions of Modin DataFrame are exctracted. Example:

```
# Extract lists of row partitions from dtrain (DMatrix object)
X_row_parts, y_row_parts = dtrain
```

2. On this step, the parameter *num_actors* is processed. The internal function *_get_num_actors()* examines the value provided by the user. In case the value isn't provided, the *num_actors* will be computed using condition that 1 actor should use maximum 2 CPUs. This condition was chosen for using maximum parallel workers with multithreaded XGBoost training (2 threads per worker will be used in this case).

Note: *num_actors* parameter is made available for public function *train()* to allow fine-tuning for obtaining the best performance in specific use cases.

---

3. *ModinXGBoostActor* objects are created.

4. Data *dtrain* is split between actors evenly. The internal function *_split_data_across_actors()* runs assigning row partitions to actors using internal function *_assign_row_partitions_to_actors()*. This function creates a dictionary in the form: *{actor_rank: ([part_i0, part_i3, ..], [0, 3, ..]), ..}*.

Note: *_assign_row_partitions_to_actors()* takes into account IP addresses of row partitions of *dtrain* data to minimize excess data transfer.

---

5. For each *ModinXGBoostActor* object `set_train_data` method is called remotely. This method runs loading row partitions in actor according to the dictionary with partitions distribution from previous step. When data is passed to the actor, the row partitions are automatically materialized (`ray.ObjectRef` -> `pandas.DataFrame`).
6. `train` method of *ModinXGBoostActor* class object is called remotely. This method runs XGBoost training on local data of actor, connects to `Rabit Tracker` for sharing training state between actors and returns dictionary with *booster* and *evaluation results*.
7. At the final stage results from actors are returned. *booster* and *evals_result* are returned using `ray.get` function from remote actor.

Prediction

1. The data is passed to *_predict()* function as a *DMatrix* object.
2. *_map_predict()* function is applied remotely for each partition of the data to make a partial prediction.
3. Result `modin.pandas.DataFrame` is created from `ray.ObjectRef` objects, obtained in the previous step.

Internal API

class modin.experimental.xgboost.xgboost_ray.**ModinXGBoostActor**(*rank*, *nthread*)

Ray actor-class runs training on the remote worker.

Parameters

- **rank** (*int*) – Rank of this actor.
- **nthread** (*int*) – Number of threads used by XGBoost in this actor.

_get_dmatrix(*X_y*, ***dmatrix_kwargs*)

Create xgboost.DMatrix from sequence of pandas.DataFrame objects.

First half of *X_y* should contains objects for *X*, second for *y*.

Parameters

- **X_y** (*list*) – List of pandas.DataFrame objects.
- ****dmatrix_kwargs** (*dict*) – Keyword parameters for `xgb.DMatrix`.

Returns A XGBoost DMatrix.

Return type xgb.DMatrix

add_eval_data(**X_y*, *eval_method*, ***dmatrix_kwargs*)

Add evaluation data for actor.

Parameters

- ***X_y** (*iterable*) – Sequence of ray.ObjectRef objects. First half of sequence is for *X* data, second for *y*. When it is passed in actor, auto-materialization of ray.ObjectRef -> pandas.DataFrame happens.
- **eval_method** (*str*) – Name of eval data.
- ****dmatrix_kwargs** (*dict*) – Keyword parameters for `xgb.DMatrix`.

set_train_data(**X_y*, *add_as_eval_method=None*, ***dmatrix_kwargs*)
Set train data for actor.

Parameters

- ***X_y** (*iterable*) – Sequence of ray.ObjectRef objects. First half of sequence is for *X* data, second for *y*. When it is passed in actor, auto-materialization of ray.ObjectRef -> pandas.DataFrame happens.
- **add_as_eval_method** (*str, optional*) – Name of eval data. Used in case when train data also used for evaluation.
- ****dmatrix_kwargs** (*dict*) – Keyword parameters for `xgb.DMatrix`.

train(*rabit_args*, *params*, **args*, ***kwargs*)
Run local XGBoost training.

Connects to Rabit Tracker environment to share training data between actors and trains XGBoost booster using *self._dtrain*.

Parameters

- **rabit_args** (*list*) – List with environment variables for Rabit Tracker.
- **params** (*dict*) – Booster params.
- ***args** (*iterable*) – Other parameters for *xgboost.train*.
- ****kwargs** (*dict*) – Other parameters for *xgboost.train*.

Returns A dictionary with trained booster and dict of evaluation results as {"booster": xgb.Booster, "history": dict}.

Return type dict

modin.experimental.xgboost.xgboost_ray.**_assign_row_partitions_to_actors**(*actors: List*, *row_partitions*, *data_for_aligning=None*)
Assign row_partitions to actors.

row_partitions will be assigned to actors according to their IPs. If distribution isn't even, partitions will be moved from actor with excess partitions to actor with lack of them.

Parameters

- **actors** (*list*) – List of used actors.
- **row_partitions** (*list*) – Row partitions of data to assign.
- **data_for_aligning** (*dict, optional*) – Data according to the order of which should be distributed *row_partitions*. Used to align y with X.

Returns Dictionary of assigned to actors partitions as {actor_rank: (partitions, order)}.

Return type dict

modin.experimental.xgboost.xgboost_ray.**_train**(*dtrain*, *params: Dict*, **args*, *num_actors=None*, *evals=()*, ***kwargs*)

Run distributed training of XGBoost model on Ray engine.

During work it evenly distributes *dtrain* between workers according to IP addresses partitions (in case of not even distribution of *dtrain* by nodes, part of partitions will be re-distributed between nodes), runs xgb.train on each worker for subset of *dtrain* and reduces training results of each worker using Rabit Context.

Parameters

- **dtrain** (*modin.experimental.DMatrix*) – Data to be trained against.
- **params** (*dict*) – Booster params.
- ***args** (*iterable*) – Other parameters for *xgboost.train*.
- **num_actors** (*int, optional*) – Number of actors for training. If unspecified, this value will be computed automatically.
- **evals** (*list of pairs (*modin.experimental.xgboost.DMatrix*, str), default: empty*) – List of validation sets for which metrics will be evaluated during training. Validation metrics will help us track the performance of the model.
- ****kwargs** (*dict*) – Other parameters are the same as *xgboost.train*.

Returns A dictionary with trained booster and dict of evaluation results as {"booster": xgboost.Booster, "history": dict}.

Return type dict

modin.experimental.xgboost.xgboost_ray.**_predict**(*booster*, *data*, ***kwargs*)

Run distributed prediction with a trained booster on Ray engine.

During execution it runs `xgb.predict` on each worker for subset of *data* and creates Modin DataFrame with prediction results.

Parameters

- **booster** (xgboost.Booster) – A trained booster.
- **data** (modin.experimental.xgboost.DMatrix) – Input data used for prediction.
- ****kwargs** (*dict*) – Other parameters are the same as for `xgboost.Booster.predict`.

Returns Modin DataFrame with prediction results.

Return type modin.pandas.DataFrame

modin.experimental.xgboost.xgboost_ray.**_get_num_actors**(*num_actors=None*)

Get number of actors to create.

Parameters **num_actors** (*int, optional*) – Desired number of actors. If is None, integer number of actors will be computed by condition 2 CPUs per 1 actor.

Returns Number of actors to create.

Return type int

modin.experimental.xgboost.xgboost_ray.**_split_data_across_actors**(*actors: List*, *set_func*, *X_parts*, *y_parts*)

Split row partitions of data between actors.

Parameters

- **actors** (*list*) – List of used actors.
- **set_func** (*callable*) – The function for setting data in actor.

- **X_parts** (*list*) – Row partitions of X data.
- **y_parts** (*list*) – Row partitions of y data.

modin.experimental.xgboost.xgboost_ray.**_map_predict**(*booster*, *part*, *columns*, *dmatrix_kwargs={}*, ***kwargs*)

Run prediction on a remote worker.

Parameters

- **booster** (*xgboost.Booster or ray.ObjectRef*) – A trained booster.
- **part** (*pandas.DataFrame or ray.ObjectRef*) – Partition of full data used for local prediction.
- **columns** (*list or ray.ObjectRef*) – Columns for the result.
- **dmatrix_kwargs** (*dict, optional*) – Keyword parameters for `xgb.DMatrix`.
- ****kwargs** (*dict*) – Other parameters are the same as for `xgboost.Booster.predict`.

Returns `ray.ObjectRef` with partial prediction.

Return type ray.ObjectRef

Storage Formats

Storage format is one of the components that form Modin's execution, it describes the type(s) of objects that are stored in the partitions of the selected Core Modin Dataframe implementation.

The base storage format in Modin is pandas. In that format, Modin Dataframe operates with partitions that hold `pandas.DataFrame` objects. Pandas is the most natural storage format since high-level DataFrame objects mirror its API, however, Modin's storage formats are not limited to the objects that conform to pandas API. There are formats that are able to store `pyarrow.Table` (*pyarrow storage format*) or even instances of SQL-like databases (OmniSci storage format) inside Modin Dataframe's partitions.

An honor of converting high-level pandas API calls to the ones that are understandable by the corresponding execution implementation belongs to the Query Compiler (QC) object.

Query Compiler

BaseQueryCompiler

Brief description

BaseQueryCompiler is an abstract class of query compiler, and sets a common interface that every other query compiler implementation in Modin must follow. The Base class contains a basic implementations for most of the interface methods, all of which *default to pandas*.

Subclassing `BaseQueryCompiler`

If you want to add new type of query compiler to Modin the new class needs to inherit from *`BaseQueryCompiler`* and implement the abstract methods:

- *`from_pandas()`* build query compiler from pandas DataFrame.
- *`from_arrow()`* build query compiler from Arrow Table.
- *`to_pandas()`* get query compiler representation as pandas DataFrame.
- *`default_to_pandas()`* do *fallback to pandas* for the passed function.
- *`finalize()`* finalize object constructing.
- *`free()`* trigger memory cleaning.

(Please refer to the code documentation to see the full documentation for these functions).

This is a minimum set of operations to ensure a new query compiler will function in the Modin architecture, and the rest of the API can safely default to the pandas implementation via the base class implementation. To add a storage format specific implementation for some of the query compiler operations, just override the corresponding method in your query compiler class.

Example

As an exercise let's define a new query compiler in *Modin*, just to see how easy it is. Usually, the query compiler routes formed queries to the underlying frame class, which submits operators to an execution engine. For the sake of simplicity and independence of this example, our execution engine will be the pandas itself.

We need to inherit a new class from *`BaseQueryCompiler`* and implement all of the abstract methods. In this case, with *pandas* as an execution engine, it's trivial:

```
from modin.core.storage_formats import BaseQueryCompiler

class DefaultToPandasQueryCompiler(BaseQueryCompiler):
    def __init__(self, pandas_df):
        self._pandas_df = pandas_df

    @classmethod
    def from_pandas(cls, df, *args, **kwargs):
        return cls(df)

    @classmethod
    def from_arrow(cls, at, *args, **kwargs):
        return cls(at.to_pandas())

    def to_pandas(self):
        return self._pandas_df.copy()

    def default_to_pandas(self, pandas_op, *args, **kwargs):
        return type(self)(pandas_op(self.to_pandas(), *args, **kwargs))

    def finalize(self):
        pass
```

(continues on next page)

(continued from previous page)

```
    def free(self):
        pass
```

All done! Now you've got a fully functional query compiler, which is ready for extensions and already can be used in Modin DataFrame:

```
import pandas
pandas_df = pandas.DataFrame({"col1": [1, 2, 2, 1], "col2": [10, 2, 3, 40]})
# Building our query compiler from pandas object
qc = DefaultToPandasQueryCompiler.from_pandas(pandas_df)

import modin.pandas as pd
# Building Modin DataFrame from newly created query compiler
modin_df = pd.DataFrame(query_compiler=qc)

# Got fully functional Modin DataFrame
>>> print(modin_df.groupby("col1").sum().reset_index())
   col1  col2
0     1    50
1     2     5
```

To be able to select this query compiler as default via `modin.config` you also need to define the combination of your query compiler and pandas engine as an execution by adding the corresponding factory. To find more information about factories, visit corresponding section of the flow documentation.

Query Compiler API

class modin.core.storage_formats.base.query_compiler.**BaseQueryCompiler**

Abstract class that handles the queries to Modin dataframes.

This class defines common query compilers API, most of the methods are already implemented and defaulting to pandas.

lazy_execution

Whether underlying execution engine is designed to be executed in a lazy mode only. If True, such QueryCompiler will be handled differently at the front-end in order to reduce execution triggering as much as possible.

Type bool

Notes

See the Abstract Methods and Fields section immediately below this for a list of requirements for subclassing this object.

abs()

Get absolute numeric value of each element.

Returns QueryCompiler with absolute numeric value of each element.

Return type *BaseQueryCompiler*

add(*other*, ***kwargs*)

Perform element-wise addition (`self + other`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (BaseQueryCompiler, *scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.
- **axis** (*{0, 1}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.
- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

add_prefix(*prefix*, *axis=1*)

Add string prefix to the index labels along specified axis.

Parameters

- **prefix** (*str*) – The string to add before each label.
- **axis** (*{0, 1}, default: 1*) – Axis to add prefix along. 0 is for index and 1 is for columns.

Returns New query compiler with updated labels.

Return type *BaseQueryCompiler*

add_suffix(*suffix*, *axis=1*)

Add string suffix to the index labels along specified axis.

Parameters

- **suffix** (*str*) – The string to add after each label.
- **axis** (*{0, 1}, default: 1*) – Axis to add suffix along. 0 is for index and 1 is for columns.

Returns New query compiler with updated labels.

Return type *BaseQueryCompiler*

all(***kwargs*)

Return whether all the elements are true, potentially over an axis.

Parameters

- **axis** (*{0, 1}, optional*) –
- **bool_only** (*bool, optional*) –
- **skipna** (*bool*) –
- **level** (*int or label*) –

- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns If axis was specified return one-column QueryCompiler with index labels of the specified axis, where each row contains boolean of whether all elements at the corresponding row or column are True. Otherwise return QueryCompiler with a single bool of whether all elements are True.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.all` for more information about parameters and output format.

any(***kwargs*)
Return whether any element is true, potentially over an axis.

Parameters

- **axis** (*{0, 1}, optional*) –
- **bool_only** (*bool, optional*) –
- **skipna** (*bool*) –
- **level** (*int or label*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns If axis was specified return one-column QueryCompiler with index labels of the specified axis, where each row contains boolean of whether any element at the corresponding row or column is True. Otherwise return QueryCompiler with a single bool of whether any element is True.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.any` for more information about parameters and output format.

apply(*func*, *axis*, **args*, ***kwargs*)
Apply passed function across given axis.

Parameters

- **func** (*callable(pandas.Series) -> scalar, str, list or dict of such*) – The function to apply to each column or row.
- **axis** (*{0, 1}*) – Target axis to apply the function along. 0 is for index, 1 is for columns.
- ***args** (*iterable*) – Positional arguments to pass to *func*.
- ****kwargs** (*dict*) – Keyword arguments to pass to *func*.

Returns

QueryCompiler that contains the results of execution and is built by the following rules:

- Labels of specified axis are the passed functions names.
- Labels of the opposite axis are preserved.

- Each element is the result of execution of *func* against corresponding row/column.

Return type *BaseQueryCompiler*

applymap(*func*)
Apply passed function elementwise.

Parameters **func** (*callable(scalar) -> scalar*) – Function to apply to each element of the QueryCompiler.

Returns Transformed QueryCompiler.

Return type *BaseQueryCompiler*

astype(*col_dtypes*, ***kwargs*)
Convert columns dtypes to given dtypes.

Parameters

- **col_dtypes** (*dict*) – Map for column names and new dtypes.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with updated dtypes.

Return type *BaseQueryCompiler*

cat_codes()
Convert underlying categories data into its codes.

Returns New QueryCompiler containing the integer codes of the underlying categories.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.cat.codes` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

clip(*lower*, *upper*, ***kwargs*)
Trim values at input threshold.

Parameters

- **lower** (*float or list-like*) –
- **upper** (*float or list-like*) –
- **axis** (*{0, 1}*) –
- **inplace** (*{False}*) – This parameter serves the compatibility purpose. Always has to be False.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler with values limited by the specified thresholds.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.clip` for more information about parameters and output format.

columnarize()
Transpose this QueryCompiler if it has a single row but multiple columns.

This method should be called for QueryCompilers representing a Series object, i.e. `self.is_series_like()` should be True.

Returns Transposed new QueryCompiler or self.

Return type *BaseQueryCompiler*

combine(*other*, ***kwargs*)
Perform column-wise combine with another QueryCompiler with passed *func*.

If axes are not equal, perform frames alignment first.

Parameters

- **other** (`BaseQueryCompiler`) – Left operand of the binary operation.
- **func** (*callable(pandas.Series, pandas.Series) -> pandas.Series*) – Function that takes two `pandas.Series` with aligned axes and returns one `pandas.Series` as resulting combination.
- **fill_value** (*float or None*) – Value to fill missing values with after frame alignment occurred.
- **overwrite** (*bool*) – If True, columns in *self* that do not exist in *other* will be overwritten with NaNs.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of combine.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.combine` for more information about parameters and output format.

combine_first(*other*, ***kwargs*)
Fill null elements of *self* with value in the same location in *other*.

If axes are not equal, perform frames alignment first.

Parameters

- **other** (`BaseQueryCompiler`) – Provided frame to use to fill null values from.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.combine_first` for more information about parameters and output format.

compare(*other*, *align_axis*, *keep_shape*, *keep_equal*)
Compare data of two QueryCompilers and highlight the difference.

Parameters

- **other** (BaseQueryCompiler) – Query compiler to compare with. Have to be the same shape and the same labeling as *self*.
- **align_axis** (*{0, 1}*) –
- **keep_shape** (*bool*) –
- **keep_equal** (*bool*) –

Returns New QueryCompiler containing the differences between *self* and passed query compiler.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.compare` for more information about parameters and output format.

concat(*axis*, *other*, ***kwargs*)
Concatenate *self* with passed query compilers along specified axis.

Parameters

- **axis** (*{0, 1}*) – Axis to concatenate along. 0 is for index and 1 is for columns.
- **other** (BaseQueryCompiler *or list of such*) – Objects to concatenate with *self*.
- **join** (*{'outer', 'inner', 'right', 'left'}, default: 'outer'*) – Type of join that will be used if indices on the other axis are different. (note: if specified, has to be passed as `join=value`).
- **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. (note: if specified, has to be passed as `ignore_index=value`).
- **sort** (*bool, default: False*) – Whether or not to sort non-concatenation axis. (note: if specified, has to be passed as `sort=value`).
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Concatenated objects.

Return type *BaseQueryCompiler*

conj(***kwargs*)
Get the complex conjugate for every element of self.

Parameters ****kwargs** (*dict*) –

Returns QueryCompiler with conjugate applied element-wise.

Return type *BaseQueryCompiler*

Notes

Please refer to `numpy.conj` for parameters description.

copy()
Make a copy of this object.

Returns Copy of self.

Return type *BaseQueryCompiler*

Notes

For copy, we don't want a situation where we modify the metadata of the copies if we end up modifying something here. We copy all of the metadata to prevent that.

corr(***kwargs*)
Compute pairwise correlation of columns, excluding NA/null values.

Parameters

- **method** (*{'pearson', 'kendall', 'spearman'} or callable(pandas.Series, pandas.Series) -> pandas.Series*) – Correlation method.
- **min_periods** (*int*) – Minimum number of observations required per pair of columns to have a valid result. If fewer than *min_periods* non-NA values are present the result will be NA.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Correlation matrix.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.corr` for more information about parameters and output format.

count(***kwargs*)
Get the number of non-NaN values for each column or row.

Parameters

- **axis** (*{{0, 1}}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the number of non-NaN values for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.count` for more information about parameters and output format.

cov(***kwargs*)
Compute pairwise covariance of columns, excluding NA/null values.

Parameters

- **min_periods** (*int*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Covariance matrix.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.cov` for more information about parameters and output format.

cummax(***kwargs*)
Get cummulative maximum for every row or column.

Parameters

- **axis** (*{0, 1}*) –
- **skipna** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler of the same shape as *self*, where each element is the maximum of all the previous values in this row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.cummax` for more information about parameters and output format.

cummin(***kwargs*)
Get cummulative minimum for every row or column.

Parameters

- **axis** (*{0, 1}*) –
- **skipna** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler of the same shape as *self*, where each element is the minimum of all the previous values in this row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.cummin` for more information about parameters and output format.

cumprod(***kwargs*)
Get cummulative product for every row or column.

Parameters

- **axis** (*{0, 1}*) –
- **skipna** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler of the same shape as *self*, where each element is the product of all the previous values in this row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.cumprod` for more information about parameters and output format.

cumsum(***kwargs*)
Get cummulative sum for every row or column.

Parameters

- **axis** (*{0, 1}*) –
- **skipna** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler of the same shape as *self*, where each element is the sum of all the previous values in this row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.cumsum` for more information about parameters and output format.

abstract default_to_pandas(*pandas_op*, **args*, ***kwargs*)
Do fallback to pandas for the passed function.

Parameters

- **pandas_op** (*callable(pandas.DataFrame) -> object*) – Function to apply to the casted to pandas frame.
- ***args** (*iterable*) – Positional arguments to pass to *pandas_op*.
- ****kwargs** (*dict*) – Key-value arguments to pass to *pandas_op*.

Returns The result of the *pandas_op*, converted back to `BaseQueryCompiler`.

Return type *BaseQueryCompiler*

delitem(*key*)
Drop *key* column.

Parameters **key** (*label*) – Column name to drop.

Returns New QueryCompiler without *key* column.

Return type *BaseQueryCompiler*

describe(***kwargs*)
Generate descriptive statistics.

Parameters

- **percentiles** (*list-like*) –
- **include** (*"all" or list of dtypes, optional*) –
- **exclude** (*list of dtypes, optional*) –
- **datetime_is_numeric** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler object containing the descriptive statistics of the underlying data.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.describe` for more information about parameters and output format.

df_update(*other*, ***kwargs*)
Update values of *self* using non-NA values of *other* at the corresponding positions.

If axes are not equal, perform frames alignment first.

Parameters

- **other** (`BaseQueryCompiler`) – Frame to grab replacement values from.
- **join** (*{"left"}*) – Specify type of join to align frames if axes are not equal (note: currently only one type of join is implemented).
- **overwrite** (*bool*) – Whether to overwrite every corresponding value of self, or only if it's NAN.
- **filter_func** (*callable(pandas.Series, pandas.Series) -> numpy.ndarray<bool>*) – Function that takes column of the self and return bool mask for values, that should be overwriten in the self frame.
- **errors** (*{"raise", "ignore"}*) – If "raise", will raise a `ValueError` if *self* and *other* both contain non-NA data in the same place.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with updated values.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.update` for more information about parameters and output format.

diff(***kwargs*)
First discrete difference of element.

Parameters

- **periods** (*int*) –
- **axis** (*{0, 1}*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler of the same shape as *self*, where each element is the difference between the corresponding value and the previous value in this row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.diff` for more information about parameters and output format.

dot(*other*, ***kwargs*)
Compute the matrix multiplication of *self* and *other*.

Parameters

- **other** (`BaseQueryCompiler` *or NumPy array*) – The other query compiler or NumPy array to matrix multiply with *self*.
- **squeeze_self** (*boolean*) – If *self* is a one-column query compiler, indicates whether it represents Series object.
- **squeeze_other** (*boolean*) – If *other* is a one-column query compiler, indicates whether it represents Series object.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns A new query compiler that contains result of the matrix multiply.

Return type *BaseQueryCompiler*

drop(*index=None*, *columns=None*)
Drop specified rows or columns.

Parameters

- **index** (*list of labels, optional*) – Labels of rows to drop.
- **columns** (*list of labels, optional*) – Labels of columns to drop.

Returns New QueryCompiler with removed data.

Return type *BaseQueryCompiler*

dropna(***kwargs*)
Remove missing values.

Parameters

- **axis** (*{0, 1}*) –

- **how** (*{"any", "all"}*) –
- **thresh** (*int, optional*) –
- **subset** (*list of labels*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with null values dropped along given axis.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.dropna` for more information about parameters and output format.

dt_ceil(*freq*, *ambiguous='raise'*, *nonexistent='raise'*)

Perform ceil operation on the underlying time-series data to the specified *freq*.

Parameters

- **freq** (*str*) –
- **ambiguous** (*{"raise", "infer", "NaT"} or bool mask, default: "raise"*) –
- **nonexistent** (*{"raise", "shift_forward", "shift_backward", "NaT"} or timedelta, default: "raise"*) –

Returns New QueryCompiler with performed ceil operation on every element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.ceil` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

dt_components()

Spread each date-time value into its components (days, hours, minutes…).

Returns

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.components` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_date()`
Get the date without timezone information for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is the date without timezone information for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.date` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_day()`
Get day component for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is day component for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.day` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_day_name(locale=None)`
Get day name for each datetime value.

Parameters **locale** (*str, optional*) –

Returns New QueryCompiler with the same shape as *self*, where each element is day name for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.day_name` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_dayofweek()`
Get integer day of week for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is integer day of week for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.dayofweek` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_dayofyear()`
Get day of year for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is day of year for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.dayofyear` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_days()`
Get days for each interval value.

Returns New QueryCompiler with the same shape as *self*, where each element is days for the corresponding interval value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.days` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_days_in_month()`
Get number of days in month for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is number of days in month for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.days_in_month` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_daysinmonth()`
Get number of days in month for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is number of days in month for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.daysinmonth` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_end_time()`
Get the timestamp of end time for each period value.

Returns New QueryCompiler with the same shape as *self*, where each element is the timestamp of end time for the corresponding period value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.end_time` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_floor`(*freq*, *ambiguous='raise'*, *nonexistent='raise'*)
Perform floor operation on the underlying time-series data to the specified *freq*.

Parameters

- **freq** (*str*) –
- **ambiguous** (*{"raise", "infer", "NaT"} or bool mask, default: "raise"*) –
- **nonexistent** (*{"raise", "shift_forward", "shift_backward", "NaT"} or timedelta, default: "raise"*) –

Returns New QueryCompiler with performed floor operation on every element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.floor` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_freq`()
Get the time frequency of the underlying time-series data.

Returns QueryCompiler containing a single value, the frequency of the data.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.freq` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_hour`()
Get hour for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is hour for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.hour` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_is_leap_year()`
Get the boolean of whether corresponding year is leap for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is the boolean of whether corresponding year is leap for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.is_leap_year` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_is_month_end()`
Get the boolean of whether the date is the last day of the month for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is the boolean of whether the date is the last day of the month for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.is_month_end` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_is_month_start()`
Get the boolean of whether the date is the first day of the month for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is the boolean of whether the date is the first day of the month for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.is_month_start` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_is_quarter_end()`
Get the boolean of whether the date is the last day of the quarter for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is the boolean of whether the date is the last day of the quarter for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.is_quarter_end` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_is_quarter_start()`
Get the boolean of whether the date is the first day of the quarter for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is the boolean of whether the date is the first day of the quarter for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.is_quarter_start` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_is_year_end()`
Get the boolean of whether the date is the last day of the year for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is the boolean of whether the date is the last day of the year for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.is_year_end` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_is_year_start()`
Get the boolean of whether the date is the first day of the year for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is the boolean of whether the date is the first day of the year for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.is_year_start` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_microsecond()`
Get microseconds component for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is microseconds component for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.microsecond` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_microseconds()`
Get microseconds component for each interval value.

Returns New QueryCompiler with the same shape as *self*, where each element is microseconds component for the corresponding interval value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.microseconds` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_minute()`
Get minute component for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is minute component for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.minute` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_month()`
Get month component for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is month component for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.month` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_month_name(locale=None)`
Get the month name for each datetime value.

Parameters **locale** (*str, optional*) –

Returns New QueryCompiler with the same shape as *self*, where each element is the month name for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.month_name` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_nanosecond()`
Get nanoseconds component for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is nanoseconds component for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.nanosecond` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_nanoseconds()`
Get nanoseconds component for each interval value.

Returns New QueryCompiler with the same shape as *self*, where each element is nanoseconds component for the corresponding interval value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.nanoseconds` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_normalize()`
Set the time component of each date-time value to midnight.

Returns New QueryCompiler containing date-time values with midnight time.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.normalize` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

dt_quarter()
Get quarter component for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is quarter component for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.quarter` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

dt_qyear()
Get the fiscal year for each period value.

Returns New QueryCompiler with the same shape as *self*, where each element is the fiscal year for the corresponding period value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.qyear` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

dt_round(*freq*, *ambiguous='raise'*, *nonexistent='raise'*)
Perform round operation on the underlying time-series data to the specified *freq*.

Parameters

- **freq** (*str*) –
- **ambiguous** (*{"raise", "infer", "NaT"} or bool mask, default: "raise"*) –
- **nonexistent** (*{"raise", "shift_forward", "shift_backward", "NaT"} or timedelta, default: "raise"*) –

Returns New QueryCompiler with performed round operation on every element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.round` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_second()`
Get seconds component for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is seconds component for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.second` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_seconds()`
Get seconds component for each interval value.

Returns New QueryCompiler with the same shape as *self*, where each element is seconds component for the corresponding interval value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.seconds` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_start_time()`
Get the timestamp of start time for each period value.

Returns New QueryCompiler with the same shape as *self*, where each element is the timestamp of start time for the corresponding period value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.start_time` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_strftime`(*date_format*)
Format underlying date-time data using specified format.

Parameters **date_format** (*str*) –

Returns New QueryCompiler containing formated date-time values.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.strftime` for more information about parameters and output format.

`dt_time`()
Get time component for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is time component for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.time` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_timetz`()
Get time component with timezone information for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is time component with timezone information for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.timetz` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_to_period`(*freq=None*)
Convert underlying data to the period at a particular frequency.

Parameters **freq** (*str, optional*) –

Returns New QueryCompiler containing period data.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.to_period` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_to_pydatetime`()
Convert underlying data to array of python native `datetime`.

Returns New QueryCompiler containing 1D array of `datetime` objects.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.to_pydatetime` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_to_pytimedelta`()
Convert underlying data to array of python native `datetime.timedelta`.

Returns New QueryCompiler containing 1D array of `datetime.timedelta`.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.to_pytimedelta` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_to_timestamp()`
Get the timestamp representation for each period value.

Returns New QueryCompiler with the same shape as *self*, where each element is the timestamp representation for the corresponding period value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.to_timestamp` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_total_seconds()`
Get duration in seconds for each interval value.

Returns New QueryCompiler with the same shape as *self*, where each element is duration in seconds for the corresponding interval value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.total_seconds` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_tz()`
Get the time-zone of the underlying time-series data.

Returns QueryCompiler containing a single value, time-zone of the data.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.tz` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_tz_convert`(*tz*)
Convert time-series data to the specified time zone.

Parameters **tz** (*str, pytz.timezone*) –

Returns New QueryCompiler containing values with converted time zone.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.tz_convert` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_tz_localize`(*tz*, *ambiguous='raise'*, *nonexistent='raise'*)
Localize tz-naive to tz-aware.

Parameters

- **tz** (*str, pytz.timezone, optional*) –
- **ambiguous** (*{"raise", "inner", "NaT"} or bool mask, default: "raise"*) –
- **nonexistent** (*{"raise", "shift_forward", "shift_backward, "NaT"} or pandas.timedelta, default: "raise"*) –

Returns New QueryCompiler containing values with localized time zone.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.tz_localize` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_week`()
Get week component for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is week component for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.week` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_weekday()`
Get integer day of week for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is integer day of week for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.weekday` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_weekofyear()`
Get week of year for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is week of year for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.weekofyear` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`dt_year()`
Get year component for each datetime value.

Returns New QueryCompiler with the same shape as *self*, where each element is year component for the corresponding datetime value.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.year` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

property dtypes
Get columns dtypes.

Returns Series with dtypes of each column.

Return type pandas.Series

eq(*other*, ***kwargs*)
Perform element-wise equality comparison (`self == other`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (*BaseQueryCompiler, scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.
- **axis** (*{{0, 1}}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.
- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

eval(*expr*, ***kwargs*)
Evaluate string expression on QueryCompiler columns.

Parameters

- **expr** (*str*) –
- ****kwargs** (*dict*) –

Returns QueryCompiler containing the result of evaluation.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.eval` for more information about parameters and output format.

explode(*column*)
Explode the given columns.

Parameters **column** (*Union[Hashable, Sequence[Hashable]]*) – The columns to explode.

Returns QueryCompiler that contains the results of execution. For each row in the input QueryCompiler, if the selected columns each contain M items, there will be M rows created by exploding the columns.

Return type *BaseQueryCompiler*

fillna(***kwargs*)
Replace NaN values using provided method.

Parameters

- **value** (*scalar or dict*) –
- **method** (*{"backfill", "bfill", "pad", "ffill", None}*) –
- **axis** (*{0, 1}*) –
- **inplace** (*{False}*) – This parameter serves the compatibility purpose. Always has to be False.
- **limit** (*int, optional*) –
- **downcast** (*dict, optional*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with all null values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.fillna` for more information about parameters and output format.

abstract finalize()
Finalize constructing the dataframe calling all deferred functions which were used to build it.

first_valid_index()
Return index label of first non-NaN/NULL value.

Returns

Return type scalar

floordiv(*other*, ***kwargs*)
Perform element-wise integer division (`self // other`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (`BaseQueryCompiler`, *scalar or array-like*) – Other operand of the binary operation.

- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.
- **axis** (*{{0, 1}}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.
- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

abstract free()
Trigger a cleanup of this object.

abstract classmethod from_arrow(*at*, *data_cls*)
Build QueryCompiler from Arrow Table.

Parameters

- **at** (*Arrow Table*) – The Arrow Table to convert from.
- **data_cls** (*type*) – *PandasDataframe* class (or its descendant) to convert to.

Returns QueryCompiler containing data from the pandas DataFrame.

Return type *BaseQueryCompiler*

abstract classmethod from_pandas(*df*, *data_cls*)
Build QueryCompiler from pandas DataFrame.

Parameters

- **df** (*pandas.DataFrame*) – The pandas DataFrame to convert from.
- **data_cls** (*type*) – *PandasDataframe* class (or its descendant) to convert to.

Returns QueryCompiler containing data from the pandas DataFrame.

Return type *BaseQueryCompiler*

ge(*other*, ***kwargs*)
Perform element-wise greater than or equal comparison (`self >= other`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (*BaseQueryCompiler, scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

get_axis(*axis*)
Return index labels of the specified axis.

Parameters **axis** (*{0, 1}*) – Axis to return labels on. 0 is for index, when 1 is for columns.

Returns

Return type pandas.Index

get_dummies(*columns*, ***kwargs*)
Convert categorical variables to dummy variables for certain columns.

Parameters

- **columns** (*label or list of such*) – Columns to convert.
- **prefix** (*str or list of such*) –
- **prefix_sep** (*str*) –
- **dummy_na** (*bool*) –
- **drop_first** (*bool*) –
- **dtype** (*dtype*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with categorical variables converted to dummy.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.get_dummies` for more information about parameters and output format.

get_index_name(*axis=0*)
Get index name of specified axis.

Parameters **axis** (*{0, 1}, default: 0*) – Axis to get index name on.

Returns Index name, None for MultiIndex.

Return type hashable

get_index_names(*axis=0*)
Get index names of specified axis.

Parameters **axis** (*{0, 1}, default: 0*) – Axis to get index names on.

Returns Index names.

Return type list

getitem_array(*key*)
Mask QueryCompiler with *key*.

Parameters **key** (BaseQueryCompiler, *np.ndarray or list of column labels*) – Boolean mask represented by QueryCompiler or `np.ndarray` of the same shape as *self*, or enumerable of columns to pick.

Returns New masked QueryCompiler.

Return type *BaseQueryCompiler*

getitem_column_array(*key*, *numeric=False*)
Get column data for target labels.

Parameters

- **key** (*list-like*) – Target labels by which to retrieve data.
- **numeric** (*bool, default: False*) – Whether or not the key passed in represents the numeric index or the named index.

Returns New QueryCompiler that contains specified columns.

Return type *BaseQueryCompiler*

getitem_row_array(*key*)
Get row data for target indices.

Parameters **key** (*list-like*) – Numeric indices of the rows to pick.

Returns New QueryCompiler that contains specified rows.

Return type *BaseQueryCompiler*

groupby_agg(*by*, *is_multi_by*, *axis*, *agg_func*, *agg_args*, *agg_kwargs*, *groupby_kwargs*, *drop=False*)
Group QueryCompiler data and apply passed aggregation function.

Parameters

- **by** (BaseQueryCompiler, *column or index label, Grouper or list of such*) – Object that determine groups.
- **is_multi_by** (*bool*) – If *by* is a QueryCompiler or list of such indicates whether it's grouping on multiple columns/rows.
- **axis** (*{0, 1}*) – Axis to group and apply aggregation function along. 0 is for index, when 1 is for columns.
- **agg_func** (*dict or callable(DataFrameGroupBy) -> DataFrame*) – Function to apply to the GroupBy object.
- **agg_args** (*dict*) – Positional arguments to pass to the *agg_func*.
- **agg_kwargs** (*dict*) – Key arguments to pass to the *agg_func*.
- **groupby_kwargs** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns QueryCompiler containing the result of groupby aggregation.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.GroupBy.aggregate` for more information about parameters and output format.

groupby_all(*by*, *axis*, *groupby_args*, *map_args*, *reduce_args=None*, *numeric_only=True*, *drop=False*)
Group QueryCompiler data and check whether all elements are True for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the boolean of whether all elements are True for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.all` for more information about parameters and output format.

groupby_any(*by*, *axis*, *groupby_args*, *map_args*, *reduce_args=None*, *numeric_only=True*, *drop=False*)
Group QueryCompiler data and check whether any element is True for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the boolean of whether there is any element which is True for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.any` for more information about parameters and output format.

groupby_count(*by*, *axis*, *groupby_args*, *map_args*, *reduce_args=None*, *numeric_only=True*, *drop=False*)
Group QueryCompiler data and count non-null values for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the number of non-null values for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.count` for more information about parameters and output format.

groupby_max(*by*, *axis*, *groupby_args*, *map_args*, *reduce_args=None*, *numeric_only=True*, *drop=False*)
Group QueryCompiler data and get the maximum value for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the maximum value for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.max` for more information about parameters and output format.

groupby_min(*by*, *axis*, *groupby_args*, *map_args*, *reduce_args=None*, *numeric_only=True*, *drop=False*)
Group QueryCompiler data and get the minimum value for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the minimum value for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.min` for more information about parameters and output format.

groupby_prod(*by*, *axis*, *groupby_args*, *map_args*, *reduce_args=None*, *numeric_only=True*, *drop=False*)
Group QueryCompiler data and compute product for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the product for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.prod` for more information about parameters and output format.

groupby_size(*by*, *axis*, *groupby_args*, *map_args*, *reduce_args=None*, *numeric_only=True*, *drop=False*)
Group QueryCompiler data and get the number of elements for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the number of elements for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.size` for more information about parameters and output format.

groupby_sum(*by*, *axis*, *groupby_args*, *map_args*, *reduce_args=None*, *numeric_only=True*, *drop=False*)
Group QueryCompiler data and compute sum for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the sum for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.sum` for more information about parameters and output format.

gt(*other*, ***kwargs*)
Perform element-wise greater than comparison (`self > other`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (*BaseQueryCompiler, scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

has_multiindex(*axis=0*)
Check if specified axis is indexed by MultiIndex.

Parameters **axis** (*{0, 1}, default: 0*) – The axis to check (0 - index, 1 - columns).

Returns True if index at specified axis is MultiIndex and False otherwise.

Return type bool

idxmax(***kwargs*)
Get position of the first occurence of the maximum for each row or column.

Parameters

- **axis** (*{0, 1}*) –
- **skipna** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains position of the maximum element for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.idxmax` for more information about parameters and output format.

idxmin(***kwargs*)
Get position of the first occurence of the minimum for each row or column.

Parameters

- **axis** (*{0, 1}*) –
- **skipna** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains position of the minimum element for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.idxmin` for more information about parameters and output format.

insert(*loc*, *column*, *value*)
Insert new column.

Parameters

- **loc** (*int*) – Insertion position.
- **column** (*label*) – Label of the new column.
- **value** (*One-column BaseQueryCompiler, 1D array or scalar*) – Data to fill new column with.

Returns QueryCompiler with new column inserted.

Return type *BaseQueryCompiler*

insert_item(*axis*, *loc*, *value*, *how='inner'*, *replace=False*)
Insert rows/columns defined by *value* at the specified position.

If frames are not aligned along specified axis, perform frames alignment first.

Parameters

- **axis** (*{0, 1}*) – Axis to insert along. 0 means insert rows, when 1 means insert columns.
- **loc** (*int*) – Position to insert *value*.
- **value** (BaseQueryCompiler) – Rows/columns to insert.
- **how** (*{"inner", "outer", "left", "right"}, default: "inner"*) – Type of join that will be used if frames are not aligned.
- **replace** (*bool, default: False*) – Whether to insert item after column/row at *loc-th* position or to replace it by *value*.

Returns New QueryCompiler with inserted values.

Return type *BaseQueryCompiler*

invert()
Apply bitwise invertion for each element of the QueryCompiler.

Returns New QueryCompiler containing bitwise invertion for each value.

Return type *BaseQueryCompiler*

is_monotonic_decreasing()
Return boolean if values in the object are monotonicly decreasing.

Returns

Return type bool

is_monotonic_increasing()
Return boolean if values in the object are monotonicly increasing.

Returns

Return type bool

`is_series_like`()
Check whether this QueryCompiler can represent `modin.pandas.Series` object.

Returns Return True if QueryCompiler has a single column or row, False otherwise.

Return type bool

`isin`(***kwargs*)
Check for each element of *self* whether it's contained in passed *values*.

Parameters

- **values** (*list-like, modin.pandas.Series, modin.pandas.DataFrame or dict*) – Values to check elements of self in.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Boolean mask for self of whether an element at the corresponding position is contained in *values*.

Return type *BaseQueryCompiler*

`isna`()
Check for each element of self whether it's NaN.

Returns Boolean mask for self of whether an element at the corresponding position is NaN.

Return type *BaseQueryCompiler*

`join`(*right*, ***kwargs*)
Join columns of another QueryCompiler.

Parameters

- **right** (BaseQueryCompiler) – QueryCompiler of the right frame to join with.
- **on** (*label or list of such*) –
- **how** (*{"left", "right", "outer", "inner"}*) –
- **lsuffix** (*str*) –
- **rsuffix** (*str*) –
- **sort** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler that contains result of the join.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.join` for more information about parameters and output format.

kurt(*axis*, *level=None*, *numeric_only=None*, *skipna=True*, ***kwargs*)
Get the unbiased kurtosis for each column or row.

Parameters

- **axis** (*{{0, 1}}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –
- **skipna** (*bool, default: True*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the unbiased kurtosis for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.kurt` for more information about parameters and output format.

last_valid_index()
Return index label of last non-NaN/NULL value.

Returns

Return type scalar

le(*other*, ***kwargs*)
Perform element-wise less than or equal comparison (`self <= other`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (*BaseQueryCompiler, scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

lt(*other*, ***kwargs*)
Perform element-wise less than comparison (`self < other`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (*BaseQueryCompiler, scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

mad(*axis*, *skipna*, *level=None*)

Get the mean absolute deviation for each column or row.

Parameters

- **axis** (*{0, 1}*) –
- **skipna** (*bool*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the mean absolute deviation for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.mad` for more information about parameters and output format.

max(***kwargs*)

Get the maximum value for each column or row.

Parameters

- **axis** (*{{0, 1}}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –
- **skipna** (*bool, default: True*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the maximum value for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.max` for more information about parameters and output format.

mean(***kwargs*)
Get the mean value for each column or row.

Parameters

- **axis** (*{{0, 1}}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –
- **skipna** (*bool, default: True*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the mean value for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.mean` for more information about parameters and output format.

median(***kwargs*)
Get the median value for each column or row.

Parameters

- **axis** (*{{0, 1}}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –
- **skipna** (*bool, default: True*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the median value for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.median` for more information about parameters and output format.

melt(**args*, ***kwargs*)
Unpivot QueryCompiler data from wide to long format.

Parameters

- **id_vars** (*list of labels, optional*) –
- **value_vars** (*list of labels, optional*) –
- **var_name** (*label*) –
- **value_name** (*label*) –
- **col_level** (*int or label*) –
- **ignore_index** (*bool*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with unpivoted data.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.melt` for more information about parameters and output format.

memory_usage(***kwargs*)
Return the memory usage of each column in bytes.

Parameters

- **index** (*bool*) –
- **deep** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of *self*, where each row contains the memory usage for the corresponding column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.memory_usage` for more information about parameters and output format.

merge(*right*, ***kwargs*)
Merge QueryCompiler objects using a database-style join.

Parameters

- **right** (`BaseQueryCompiler`) – QueryCompiler of the right frame to merge with.
- **how** (*{"left", "right", "outer", "inner", "cross"}*) –

- **on** (*label or list of such*) –
- **left_on** (*label or list of such*) –
- **right_on** (*label or list of such*) –
- **left_index** (*bool*) –
- **right_index** (*bool*) –
- **sort** (*bool*) –
- **suffixes** (*list-like*) –
- **copy** (*bool*) –
- **indicator** (*bool or str*) –
- **validate** (*str*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler that contains result of the merge.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.merge` for more information about parameters and output format.

min(***kwargs*)

Get the minimum value for each column or row.

Parameters

- **axis** (*{{0, 1}}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –
- **skipna** (*bool, default: True*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the minimum value for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.min` for more information about parameters and output format.

mod(*other*, ***kwargs*)

Perform element-wise modulo (`self % other`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (BaseQueryCompiler, *scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.
- **axis** (*{0, 1}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.
- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

mode(***kwargs*)

Get the modes for every column or row.

Parameters

- **axis** (*{0, 1}*) –
- **numeric_only** (*bool*) –
- **dropna** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with modes calculated alogn given axis.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.mode` for more information about parameters and output format.

mul(*other*, ***kwargs*)

Perform element-wise multiplication (`self * other`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (BaseQueryCompiler, *scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.

- **axis** (*{{0, 1}}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.
- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

ne(*other*, ***kwargs*)

Perform element-wise not equal comparison (`self != other`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (*BaseQueryCompiler, scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

negative(***kwargs*)

Change the sign for every value of self.

Parameters ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

Return type *BaseQueryCompiler*

Notes

Be aware, that all QueryCompiler values have to be numeric.

nlargest(*n=5*, *columns=None*, *keep='first'*)

Return the first *n* rows ordered by *columns* in descending order.

Parameters

- **n** (*int, default: 5*) –
- **columns** (*list of labels, optional*) – Column labels to order by. (note: this parameter can be omitted only for a single-column query compilers representing Series object, otherwise *columns* has to be specified).
- **keep** (*{"first", "last", "all"}, default: "first"*) –

Returns

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.nlargest` for more information about parameters and output format.

notna()
Check for each element of *self* whether it's existing (non-missing) value.

Returns Boolean mask for *self* of whether an element at the corresponding position is not NaN.

Return type *BaseQueryCompiler*

nsmallest(*n=5*, *columns=None*, *keep='first'*)
Return the first *n* rows ordered by *columns* in ascending order.

Parameters

- **n** (*int, default: 5*) –
- **columns** (*list of labels, optional*) – Column labels to order by. (note: this parameter can be omitted only for a single-column query compilers representing Series object, otherwise *columns* has to be specified).
- **keep** (*{"first", "last", "all"}, default: "first"*) –

Returns

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.nsmallest` for more information about parameters and output format.

nunique(***kwargs*)
Get the number of unique values for each column or row.

Parameters

- **axis** (*{0, 1}*) –
- **dropna** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the number of unique values for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.nunique` for more information about parameters and output format.

pivot(*index*, *columns*, *values*)
Produce pivot table based on column values.

Parameters

- **index** (*label or list of such, pandas.Index, optional*) –

- **columns** (*label or list of such*) –
- **values** (*label or list of such, optional*) –

Returns New QueryCompiler containing pivot table.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.pivot` for more information about parameters and output format.

pivot_table(*index*, *values*, *columns*, *aggfunc*, *fill_value*, *margins*, *dropna*, *margins_name*, *observed*, *sort*)
Create a spreadsheet-style pivot table from underlying data.

Parameters

- **index** (*label, pandas.Grouper, array or list of such*) –
- **values** (*label, optional*) –
- **columns** (*column, pandas.Grouper, array or list of such*) –
- **aggfunc** (*callable(pandas.Series) -> scalar, dict of list of such*) –
- **fill_value** (*scalar, optional*) –
- **margins** (*bool*) –
- **dropna** (*bool*) –
- **margins_name** (*str*) –
- **observed** (*bool*) –
- **sort** (*bool*) –

Returns

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.pivot_table` for more information about parameters and output format.

pow(*other*, ***kwargs*)
Perform element-wise exponential power (`self ** other`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (BaseQueryCompiler, *scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.

- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.
- **axis** (*{{0, 1}}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.
- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

prod(***kwargs*)

Get the production for each column or row.

Parameters

- **axis** (*{0, 1}*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the production for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.prod` for more information about parameters and output format.

prod_min_count(***kwargs*)

Get the production for each column or row.

Parameters

- **axis** (*{0, 1}*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the production for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.prod` for more information about parameters and output format.

quantile_for_list_of_values(***kwargs*)

Get the value at the given quantile for each column or row.

Parameters

- **q** (*list-like*) –
- **axis** (*{0, 1}*) –
- **numeric_only** (*bool*) –

- **interpolation** (*{"linear", "lower", "higher", "midpoint", "nearest"}*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the value at the given quantile for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.quantile` for more information about parameters and output format.

quantile_for_single_value(***kwargs*)

Get the value at the given quantile for each column or row.

Parameters

- **q** (*float*) –
- **axis** (*{0, 1}*) –
- **numeric_only** (*bool*) –
- **interpolation** (*{"linear", "lower", "higher", "midpoint", "nearest"}*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the value at the given quantile for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.quantile` for more information about parameters and output format.

query(*expr*, ***kwargs*)

Query columns of the QueryCompiler with a boolean expression.

Parameters

- **expr** (*str*) –
- ****kwargs** (*dict*) –

Returns New QueryCompiler containing the rows where the boolean expression is satisfied.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.query` for more information about parameters and output format.

rank(***kwargs*)
Compute numerical rank along the specified axis.

By default, equal values are assigned a rank that is the average of the ranks of those values, this behaviour can be changed via *method* parameter.

Parameters

- **axis** (*{0, 1}*) –
- **method** (*{"average", "min", "max", "first", "dense"}*) –
- **numeric_only** (*bool*) –
- **na_option** (*{"keep", "top", "bottom"}*) –
- **ascending** (*bool*) –
- **pct** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler of the same shape as *self*, where each element is the numerical rank of the corresponding value along row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.rank` for more information about parameters and output format.

reindex(*axis*, *labels*, ***kwargs*)
Align QueryCompiler data with a new index along specified axis.

Parameters

- **axis** (*{0, 1}*) – Axis to align labels along. 0 is for index, 1 is for columns.
- **labels** (*list-like*) – Index-labels to align with.
- **method** (*{None, "backfill"/"bfill", "pad"/"ffill", "nearest"}*) – Method to use for filling holes in reindexed frame.
- **fill_value** (*scalar*) – Value to use for missing values in the resulted frame.
- **limit** (*int*) –
- **tolerance** (*int*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler with aligned axis.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.reindex` for more information about parameters and output format.

repeat(*repeats*)
Repeat each element of one-column QueryCompiler given number of times.

Parameters **repeats** (*int or array of ints*) – The number of repetitions for each element. This should be a non-negative integer. Repeating 0 times will return an empty QueryCompiler.

Returns New QueryCompiler with repeated elements.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.repeat` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

replace(***kwargs*)
Replace values given in *to_replace* by *value*.

Parameters

- **to_replace** (*scalar, list-like, regex, modin.pandas.Series, or None*) –
- **value** (*scalar, list-like, regex or dict*) –
- **inplace** (*{False}*) – This parameter serves the compatibility purpose. Always has to be False.
- **limit** (*int or None*) –
- **regex** (bool or same types as *to_replace*) –
- **method** (*{"pad", "ffill", "bfill", None}*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler with all *to_replace* values replaced by *value*.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.replace` for more information about parameters and output format.

resample_agg_df(*resample_args*, *func*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and apply passed aggregation function for each group over the specified axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **func** (*str, dict, callable(pandas.Series) -> scalar, or list of such*) –
- ***args** (*iterable*) – Positional arguments to pass to the aggregation function.
- ****kwargs** (*dict*) – Keyword arguments to pass to the aggregation function.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are a MultiIndex, where first level contains preserved labels of this axis and the second level is the function names.
- Each element of QueryCompiler is the result of corresponding function for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.agg` for more information about parameters and output format.

resample_agg_ser(*resample_args*, *func*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and apply passed aggregation function in a one-column query compiler for each group over the specified axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **func** (*str, dict, callable(pandas.Series) -> scalar, or list of such*) –
- ***args** (*iterable*) – Positional arguments to pass to the aggregation function.
- ****kwargs** (*dict*) – Keyword arguments to pass to the aggregation function.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are a MultiIndex, where first level contains preserved labels of this axis and the second level is the function names.
- Each element of QueryCompiler is the result of corresponding function for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.agg` for more information about parameters and output format.

> **Warning:** This method duplicates logic of `resample_agg_df` and will be removed soon.

resample_app_df(*resample_args*, *func*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and apply passed aggregation function for each group over the specified axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **func** (*str, dict, callable(pandas.Series) -> scalar, or list of such*) –
- ***args** (*iterable*) – Positional arguments to pass to the aggregation function.
- ****kwargs** (*dict*) – Keyword arguments to pass to the aggregation function.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are a MultiIndex, where first level contains preserved labels of this axis and the second level is the function names.
- Each element of QueryCompiler is the result of corresponding function for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.apply` for more information about parameters and output format.

> **Warning:** This method duplicates logic of `resample_agg_df` and will be removed soon.

resample_app_ser(*resample_args*, *func*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and apply passed aggregation function in a one-column query compiler for each group over the specified axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.

- **func** (*str, dict, callable(pandas.Series) -> scalar, or list of such*) –
- ***args** (*iterable*) – Positional arguments to pass to the aggregation function.
- ****kwargs** (*dict*) – Keyword arguments to pass to the aggregation function.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are a MultiIndex, where first level contains preserved labels of this axis and the second level is the function names.
- Each element of QueryCompiler is the result of corresponding function for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.apply` for more information about parameters and output format.

Warning: This method duplicates logic of `resample_agg_df` and will be removed soon.

resample_asfreq(*resample_args*, *fill_value*)

Resample time-series data and get the values at the new frequency.

Group data into intervals by time-series row/column with a specified frequency and get values at the new frequency.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **fill_value** (*scalar*) –

Returns New QueryCompiler containing values at the specified frequency.

Return type *BaseQueryCompiler*

resample_backfill(*resample_args*, *limit*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using back-fill method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **limit** (*int*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.backfill` for more information about parameters and output format.

resample_bfill(*resample_args*, *limit*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using back-fill method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **limit** (*int*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.bfill` for more information about parameters and output format.

resample_count(*resample_args*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute number of non-NA values for each group.

Parameters **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the number of non-NA values for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.count` for more information about parameters and output format.

resample_ffill(*resample_args*, *limit*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using forward-fill method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **limit** (*int*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.ffill` for more information about parameters and output format.

resample_fillna(*resample_args*, *method*, *limit*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using specified method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **method** (*str*) –
- **limit** (*int*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.fillna` for more information about parameters and output format.

resample_first(*resample_args*, *_method*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute first element for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the first element for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.first` for more information about parameters and output format.

resample_get_group(*resample_args*, *name*, *obj*)
Resample time-series data and get the specified group.

Group data into intervals by time-series row/column with a specified frequency and get the values of the specified group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **name** (*object*) –
- **obj** (*modin.pandas.DataFrame, optional*) –

Returns New QueryCompiler containing the values from the specified group.

Return type *BaseQueryCompiler*

resample_interpolate(*resample_args*, *method*, *axis*, *limit*, *inplace*, *limit_direction*, *limit_area*, *downcast*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using specified interpolation method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **method** (*str*) –
- **axis** (*{0, 1}*) –
- **limit** (*int*) –
- **inplace** (*{False}*) – This parameter serves the compatibility purpose. Always has to be False.
- **limit_direction** (*{"forward", "backward", "both"}*) –
- **limit_area** (*{None, "inside", "outside"}*) –
- **downcast** (*str, optional*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.interpolate` for more information about parameters and output format.

resample_last(*resample_args*, *_method*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute last element for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.

- Each element of QueryCompiler is the last element for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.last` for more information about parameters and output format.

resample_max(*resample_args*, *_method*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute maximum value for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the maximum value for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.max` for more information about parameters and output format.

resample_mean(*resample_args*, *_method*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute mean value for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the mean value for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.mean` for more information about parameters and output format.

resample_median(*resample_args*, *_method*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute median value for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the median value for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.median` for more information about parameters and output format.

resample_min(*resample_args*, *_method*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute minimum value for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the minimum value for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.min` for more information about parameters and output format.

resample_nearest(*resample_args*, *limit*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using 'nearest' method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **limit** (*int*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.nearest` for more information about parameters and output format.

resample_nunique(*resample_args*, *_method*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute number of unique values for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the number of unique values for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.nunique` for more information about parameters and output format.

resample_ohlc_df(*resample_args*, *_method*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute open, high, low and close values for each group over the specified axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Positional arguments to pass to the aggregation function.
- ****kwargs** (*dict*) – Keyword arguments to pass to the aggregation function.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are a MultiIndex, where first level contains preserved labels of this axis and the second level is the labels of columns containing computed values.
- Each element of QueryCompiler is the result of corresponding function for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.ohlc` for more information about parameters and output format.

resample_ohlc_ser(*resample_args*, *_method*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute open, high, low and close values for each group over the specified axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Positional arguments to pass to the aggregation function.
- ****kwargs** (*dict*) – Keyword arguments to pass to the aggregation function.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are a MultiIndex, where first level contains preserved labels of this axis and the second level is the labels of columns containing computed values.
- Each element of QueryCompiler is the result of corresponding function for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.ohlc` for more information about parameters and output format.

resample_pad(*resample_args*, *limit*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using 'pad' method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **limit** (*int*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.pad` for more information about parameters and output format.

resample_pipe(*resample_args*, *func*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency, build equivalent `pandas.Resampler` object and apply passed function to it.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **func** (*callable(pandas.Resampler) -> object or tuple(callable, str)*) –
- ***args** (*iterable*) – Positional arguments to pass to function.
- ****kwargs** (*dict*) – Keyword arguments to pass to function.

Returns New QueryCompiler containing the result of passed function.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Resampler.pipe` for more information about parameters and output format.

resample_prod(*resample_args*, *_method*, *min_count*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute product for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- **min_count** (*int*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the product for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.prod` for more information about parameters and output format.

resample_quantile(*resample_args*, *q*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute quantile for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **q** (*float*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the quantile for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.quantile` for more information about parameters and output format.

resample_sem(*resample_args*, *ddof=1*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute standart error of the mean for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **ddof** (*int, default: 1*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)

- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the standart error of the mean for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.sem` for more information about parameters and output format.

`resample_size`(*resample_args*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute number of elements in a group for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the number of elements in a group for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.size` for more information about parameters and output format.

`resample_std`(*resample_args*, *ddof*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute standart deviation for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **ddof** (*int*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the standart deviation for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.std` for more information about parameters and output format.

resample_sum(*resample_args*, *_method*, *min_count*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute sum for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- **min_count** (*int*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the sum for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.sum` for more information about parameters and output format.

resample_transform(*resample_args*, *arg*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and call passed function on each group. In contrast to `resample_app_df` apply function to the whole group, instead of a single axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **arg** (*callable(pandas.DataFrame) -> pandas.Series*) –
- ***args** (*iterable*) – Positional arguments to pass to function.
- ****kwargs** (*dict*) – Keyword arguments to pass to function.

Returns New QueryCompiler containing the result of passed function.

Return type *BaseQueryCompiler*

resample_var(*resample_args*, *ddof*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute variance for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **ddof** (*int*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the variance for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.var` for more information about parameters and output format.

reset_index(***kwargs*)

Reset the index, or a level of it.

Parameters

- **drop** (*bool*) – Whether to drop the reset index or insert it at the beginning of the frame.
- **level** (*int or label, optional*) – Level to remove from index. Removes all levels by default.
- **col_level** (*int or label*) – If the columns have multiple levels, determines which level the labels are inserted into.

- **col_fill** (*label*) – If the columns have multiple levels, determines how the other levels are named.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler with reset index.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.reset_index` for more information about parameters and output format.

rfloordiv(*other*, ***kwargs*)
Perform element-wise integer division (`other // self`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (*BaseQueryCompiler, scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.
- **axis** (*{0, 1}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.
- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

rmod(*other*, ***kwargs*)
Perform element-wise modulo (`other % self`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (*BaseQueryCompiler, scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.
- **axis** (*{0, 1}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.

- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

rolling_aggregate(*rolling_args*, *func*, **args*, ***kwargs*)

Create rolling window and apply specified functions for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- **func** (*str, dict, callable(pandas.Series) -> scalar, or list of such*) –
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing the result of passed functions for each window, built by the following rulles:

- Labels on the specified axis are preserved.
- Labels on the opposit of specified axis are MultiIndex, where first level contains preserved labels of this axis and the second level has the function names.
- Each element of QueryCompiler is the result of corresponding function for the corresponding window and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.aggregate` for more information about parameters and output format.

rolling_apply(*rolling_args*, *func*, *raw=False*, *engine=None*, *engine_kwargs=None*, *args=None*, *kwargs=None*)

Create rolling window and apply specified function for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- **func** (*callable(pandas.Series) -> scalar*) –
- **raw** (*bool, default: False*) –
- **engine** (*None, default: None*) – This parameters serves the compatibility purpose. Always has to be None.
- **engine_kwargs** (*None, default: None*) – This parameters serves the compatibility purpose. Always has to be None.
- **args** (*tuple, optional*) –

- **kwargs** (*dict, optional*) –

Returns

New QueryCompiler containing the result of passed function for each window, built by the following rulles:

- Labels on the specified axis are preserved.
- Labels on the opposit of specified axis are MultiIndex, where first level contains preserved labels of this axis and the second level has the function names.
- Each element of QueryCompiler is the result of corresponding function for the corresponding window and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.apply` for more information about parameters and output format.

Warning: This method duplicates logic of `rolling_aggregate` and will be removed soon.

rolling_corr(*rolling_args*, *other=None*, *pairwise=None*, **args*, ***kwargs*)
Create rolling window and compute correlation for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- **other** (*modin.pandas.Series, modin.pandas.DataFrame, list-like, optional*) –
- **pairwise** (*bool, optional*) –
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing correlation for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the correlation for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.corr` for more information about parameters and output format.

rolling_count(*rolling_args*)
Create rolling window and compute number of non-NA values for each window.

Parameters **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.

Returns

New QueryCompiler containing number of non-NA values for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the number of non-NA values for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.count` for more information about parameters and output format.

rolling_cov(*rolling_args*, *other=None*, *pairwise=None*, *ddof=1*, ***kwargs*)
Create rolling window and compute covariance for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- **other** (*modin.pandas.Series, modin.pandas.DataFrame, list-like, optional*) –
- **pairwise** (*bool, optional*) –
- **ddof** (*int, default: 1*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing covariance for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the covariance for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.cov` for more information about parameters and output format.

rolling_kurt(*rolling_args*, ***kwargs*)
Create rolling window and compute unbiased kurtosis for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing unbiased kurtosis for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the unbiased kurtosis for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.kurt` for more information about parameters and output format.

rolling_max(*rolling_args*, **args*, ***kwargs*)
Create rolling window and compute maximum value for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing maximum value for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the maximum value for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.max` for more information about parameters and output format.

rolling_mean(*rolling_args*, **args*, ***kwargs*)
Create rolling window and compute mean value for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.

- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing mean value for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the mean value for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.mean` for more information about parameters and output format.

rolling_median(*rolling_args*, ***kwargs*)
Create rolling window and compute median value for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing median value for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the median value for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.median` for more information about parameters and output format.

rolling_min(*rolling_args*, **args*, ***kwargs*)
Create rolling window and compute minimum value for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing minimum value for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the minimum value for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.min` for more information about parameters and output format.

rolling_quantile(*rolling_args*, *quantile*, *interpolation='linear'*, ***kwargs*)
Create rolling window and compute quantile for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- **quantile** (*float*) –
- **interpolation** (*{'linear', 'lower', 'higher', 'midpoint', 'nearest'}, default: 'linear'*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing quantile for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the quantile for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.quantile` for more information about parameters and output format.

rolling_skew(*rolling_args*, ***kwargs*)
Create rolling window and compute unbiased skewness for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing unbiased skewness for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the unbiased skewness for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.skew` for more information about parameters and output format.

rolling_std(*rolling_args*, *ddof=1*, **args*, ***kwargs*)
Create rolling window and compute standart deviation for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- **ddof** (*int, default: 1*) –
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing standart deviation for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the standart deviation for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.std` for more information about parameters and output format.

rolling_sum(*rolling_args*, **args*, ***kwargs*)
Create rolling window and compute sum for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing sum for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the sum for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.sum` for more information about parameters and output format.

rolling_var(*rolling_args*, *ddof=1*, **args*, ***kwargs*)
Create rolling window and compute variance for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- **ddof** (*int, default: 1*) –
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing variance for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the variance for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.var` for more information about parameters and output format.

round(***kwargs*)
Round every numeric value up to specified number of decimals.

Parameters

- **decimals** (*int or list-like*) – Number of decimals to round each column to.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler with rounded values.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.round` for more information about parameters and output format.

rpow(*other*, ***kwargs*)
Perform element-wise exponential power (`other ** self`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (`BaseQueryCompiler`, *scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.

- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.
- **axis** (*{{0, 1}}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.
- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

rsub(*other*, ***kwargs*)

Perform element-wise substraction (`other - self`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (BaseQueryCompiler, *scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.
- **axis** (*{{0, 1}}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.
- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

rtruediv(*other*, ***kwargs*)

Perform element-wise division (`other / self`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (BaseQueryCompiler, *scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.
- **axis** (*{{0, 1}}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.

- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

searchsorted(***kwargs*)

Find positions in a sorted *self* where *value* should be inserted to maintain order.

Parameters

- **value** (*list-like*) –
- **side** (*{"left", "right"}*) –
- **sorter** (*list-like, optional*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler which contains indices to insert.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.searchsorted` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

sem(***kwargs*)

Get the standard deviation of the mean for each column or row.

Parameters

- **axis** (*{{0, 1}}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –
- **skipna** (*bool, default: True*) –
- **ddof** (*int*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the standard deviation of the mean for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.sem` for more information about parameters and output format.

series_update(*other*, ***kwargs*)
Update values of *self* using values of *other* at the corresponding indices.

Parameters

- **other** (*BaseQueryCompiler*) – One-column query compiler with updated values.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with updated values.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.update` for more information about parameters and output format.

series_view(***kwargs*)
Reinterpret underlying data with new dtype.

Parameters

- **dtype** (*dtype*) – Data type to reinterpret underlying data with.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler of the same data in memory, with reinterpreted values.

Return type *BaseQueryCompiler*

Notes

- Be aware, that if this method do fallback to pandas, then newly created QueryCompiler will be the copy of the original data.
- Please refer to `modin.pandas.Series.view` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

set_index_from_columns(*keys: List[Hashable]*, *drop: bool = True*, *append: bool = False*)
Create new row labels from a list of columns.

Parameters

- **keys** (*list of hashable*) – The list of column names that will become the new index.
- **drop** (*bool, default: True*) – Whether or not to drop the columns provided in the *keys* argument.
- **append** (*bool, default: True*) – Whether or not to add the columns in *keys* as new levels appended to the existing index.

Returns A new QueryCompiler with updated index.

Return type *BaseQueryCompiler*

set_index_name(*name*, *axis=0*)
Set index name for the specified axis.

Parameters

- **name** (*hashable*) – New index name.
- **axis** (*{0, 1}, default: 0*) – Axis to set name along.

set_index_names(*names*, *axis=0*)
Set index names for the specified axis.

Parameters

- **names** (*list*) – New index names.
- **axis** (*{0, 1}, default: 0*) – Axis to set names along.

setitem(*axis*, *key*, *value*)
Set the row/column defined by *key* to the *value* provided.

Parameters

- **axis** (*{0, 1}*) – Axis to set *value* along. 0 means set row, 1 means set column.
- **key** (*label*) – Row/column label to set *value* in.
- **value** (*BaseQueryCompiler, list-like or scalar*) – Define new row/column value.

Returns New QueryCompiler with updated *key* value.

Return type *BaseQueryCompiler*

skew(***kwargs*)
Get the unbiased skew for each column or row.

Parameters

- **axis** (*{{0, 1}}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –
- **skipna** (*bool, default: True*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the unbiased skew for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.skew` for more information about parameters and output format.

sort_columns_by_row_values(*rows*, *ascending=True*, ***kwargs*)
Reorder the columns based on the lexicographic order of the given rows.

Parameters

- **rows** (*label or list of labels*) – The row or rows to sort by.
- **ascending** (*bool, default: True*) – Sort in ascending order (True) or descending order (False).
- **kind** (*{"quicksort", "mergesort", "heapsort"}*) –
- **na_position** (*{"first", "last"}*) –
- **ignore_index** (*bool*) –
- **key** (*callable(pandas.Index) -> pandas.Index, optional*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler that contains result of the sort.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.sort_values` for more information about parameters and output format.

sort_index(***kwargs*)
Sort data by index or column labels.

Parameters

- **axis** (*{0, 1}*) –
- **level** (*int, label or list of such*) –
- **ascending** (*bool*) –
- **inplace** (*bool*) –
- **kind** (*{"quicksort", "mergesort", "heapsort"}*) –
- **na_position** (*{"first", "last"}*) –
- **sort_remaining** (*bool*) –
- **ignore_index** (*bool*) –
- **key** (*callable(pandas.Index) -> pandas.Index, optional*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler containing the data sorted by columns or indices.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.sort_index` for more information about parameters and output format.

sort_rows_by_column_values(*columns*, *ascending=True*, ***kwargs*)
Reorder the rows based on the lexicographic order of the given columns.

Parameters

- **columns** (*label or list of labels*) – The column or columns to sort by.
- **ascending** (*bool, default: True*) – Sort in ascending order (True) or descending order (False).
- **kind** (*{"quicksort", "mergesort", "heapsort"}*) –
- **na_position** (*{"first", "last"}*) –
- **ignore_index** (*bool*) –
- **key** (*callable(pandas.Index) -> pandas.Index, optional*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler that contains result of the sort.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.sort_values` for more information about parameters and output format.

stack(*level*, *dropna*)
Stack the prescribed level(s) from columns to index.

Parameters

- **level** (*int or label*) –
- **dropna** (*bool*) –

Returns

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.stack` for more information about parameters and output format.

std(***kwargs*)
Get the standard deviation for each column or row.

Parameters

- **axis** (*{0, 1}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –

- **skipna** (*bool, default: True*) –
- **ddof** (*int*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the standard deviation for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.std` for more information about parameters and output format.

str___getitem__(*key*)

Apply "__getitem__" function to each string value in QueryCompiler.

Parameters **key** (*object*) –

Returns New QueryCompiler containing the result of execution of the "__getitem__" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.__getitem__` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_capitalize()

Apply "capitalize" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "capitalize" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.capitalize` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_center(*width*, *fillchar=' '*)

Apply "center" function to each string value in QueryCompiler.

Parameters

- **width** (*int*) –
- **fillchar** (*str, default: ' '*) –

Returns New QueryCompiler containing the result of execution of the "center" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.center` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

`str_contains`(*pat*, *case=True*, *flags=0*, *na=nan*, *regex=True*)

Apply "contains" function to each string value in QueryCompiler.

Parameters

- **pat** (*str*) –
- **case** (*bool, default: True*) –
- **flags** (*int, default: 0*) –
- **na** (*object, default: np.NaN*) –
- **regex** (*bool, default: True*) –

Returns New QueryCompiler containing the result of execution of the "contains" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.contains` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

`str_count`(*pat*, *flags=0*, ***kwargs*)

Apply "count" function to each string value in QueryCompiler.

Parameters

- **pat** (*str*) –
- **flags** (*int, default: 0*) –
- ****kwargs** (*dict*) –

Returns New QueryCompiler containing the result of execution of the "count" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.count` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

str_endswith(*pat*, *na=nan*)
Apply "endswith" function to each string value in QueryCompiler.

Parameters

- **pat** (*str*) –
- **na** (*object, default: np.NaN*) –

Returns New QueryCompiler containing the result of execution of the "endswith" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.endswith` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

str_find(*sub*, *start=0*, *end=None*)
Apply "find" function to each string value in QueryCompiler.

Parameters

- **sub** (*str*) –
- **start** (*int, default: 0*) –
- **end** (*int, optional*) –

Returns New QueryCompiler containing the result of execution of the "find" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.find` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

str_findall(*pat*, *flags=0*, ***kwargs*)
Apply "findall" function to each string value in QueryCompiler.

Parameters

- **pat** (*str*) –
- **flags** (*int, default: 0*) –
- ****kwargs** (*dict*) –

Returns New QueryCompiler containing the result of execution of the "findall" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.findall` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_get(*i*)

Apply "get" function to each string value in QueryCompiler.

Parameters **i** (*int*) –

Returns New QueryCompiler containing the result of execution of the "get" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.get` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_index(*sub*, *start=0*, *end=None*)

Apply "index" function to each string value in QueryCompiler.

Parameters

- **sub** (*str*) –
- **start** (*int, default: 0*) –
- **end** (*int, optional*) –

Returns New QueryCompiler containing the result of execution of the "index" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.index` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_isalnum()`
Apply "isalnum" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "isalnum" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.isalnum` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_isalpha()`
Apply "isalpha" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "isalpha" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.isalpha` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_isdecimal()`
Apply "isdecimal" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "isdecimal" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.isdecimal` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_isdigit()`
Apply "isdigit" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "isdigit" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.isdigit` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_islower()`
Apply "islower" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "islower" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.islower` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_isnumeric()`
Apply "isnumeric" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "isnumeric" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.isnumeric` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_isspace()`
Apply "isspace" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "isspace" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.isspace` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_istitle()`
Apply "istitle" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "istitle" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.istitle` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_isupper()`
Apply "isupper" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "isupper" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.isupper` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_join(*sep*)
Apply "join" function to each string value in QueryCompiler.

Parameters **sep** (*str*) –

Returns New QueryCompiler containing the result of execution of the "join" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.join` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_len()
Apply "len" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "len" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.len` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_ljust(*width*, *fillchar=' '*)
Apply "ljust" function to each string value in QueryCompiler.

Parameters

- **width** (*int*) –
- **fillchar** (*str, default: ' '*) –

Returns New QueryCompiler containing the result of execution of the "ljust" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.ljust` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_lower()`
Apply "lower" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "lower" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.lower` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_lstrip`(*to_strip=None*)
Apply "lstrip" function to each string value in QueryCompiler.

Parameters **to_strip** (*str, optional*) –

Returns New QueryCompiler containing the result of execution of the "lstrip" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.lstrip` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_match`(*pat*, *case=True*, *flags=0*, *na=nan*)
Apply "match" function to each string value in QueryCompiler.

Parameters

- **pat** (*str*) –
- **case** (*bool, default: True*) –
- **flags** (*int, default: 0*) –
- **na** (*object, default: np.NaN*) –

Returns New QueryCompiler containing the result of execution of the "match" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.match` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_normalize(*form*)

Apply "normalize" function to each string value in QueryCompiler.

Parameters **form** (*{'NFC', 'NFKC', 'NFD', 'NFKD'}*) –

Returns New QueryCompiler containing the result of execution of the "normalize" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.normalize` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_pad(*width*, *side='left'*, *fillchar=' '*)

Apply "pad" function to each string value in QueryCompiler.

Parameters

- **width** (*int*) –
- **side** (*{'left', 'right', 'both'}, default: 'left'*) –
- **fillchar** (*str, default: ' '*) –

Returns New QueryCompiler containing the result of execution of the "pad" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.pad` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_partition(*sep=' '*, *expand=True*)

Apply "partition" function to each string value in QueryCompiler.

Parameters

- **sep** (*str, default: ' '*) –
- **expand** (*bool, default: True*) –

Returns New QueryCompiler containing the result of execution of the "partition" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.partition` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

`str_repeat`(*repeats*)

Apply "repeat" function to each string value in QueryCompiler.

Parameters **repeats** (*int*) –

Returns New QueryCompiler containing the result of execution of the "repeat" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.repeat` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

`str_replace`(*pat*, *repl*, *n=- 1*, *case=None*, *flags=0*, *regex=True*)

Apply "replace" function to each string value in QueryCompiler.

Parameters

- **pat** (*str*) –
- **repl** (*str or callable*) –
- **n** (*int, default: -1*) –
- **case** (*bool, optional*) –
- **flags** (*int, default: 0*) –
- **regex** (*bool, default: True*) –

Returns New QueryCompiler containing the result of execution of the "replace" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.replace` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_rfind(*sub*, *start=0*, *end=None*)
Apply "rfind" function to each string value in QueryCompiler.

Parameters

- **sub** (*str*) –
- **start** (*int, default: 0*) –
- **end** (*int, optional*) –

Returns New QueryCompiler containing the result of execution of the "rfind" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.rfind` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_rindex(*sub*, *start=0*, *end=None*)
Apply "rindex" function to each string value in QueryCompiler.

Parameters

- **sub** (*str*) –
- **start** (*int, default: 0*) –
- **end** (*int, optional*) –

Returns New QueryCompiler containing the result of execution of the "rindex" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.rindex` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_rjust(*width*, *fillchar=' '*)
Apply "rjust" function to each string value in QueryCompiler.

Parameters

- **width** (*int*) –
- **fillchar** (*str, default: ' '*) –

Returns New QueryCompiler containing the result of execution of the "rjust" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.rjust` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_rpartition(*sep=' '*, *expand=True*)
Apply "rpartition" function to each string value in QueryCompiler.

Parameters

- **sep** (*str, default: ' '*) –
- **expand** (*bool, default: True*) –

Returns New QueryCompiler containing the result of execution of the "rpartition" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.rpartition` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_rsplit(*pat=None*, *n=-1*, *expand=False*)
Apply "rsplit" function to each string value in QueryCompiler.

Parameters

- **pat** (*str, optional*) –
- **n** (*int, default: -1*) –
- **expand** (*bool, default: False*) –

Returns New QueryCompiler containing the result of execution of the "rsplit" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.rsplit` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

str_rstrip(*to_strip=None*)

Apply "rstrip" function to each string value in QueryCompiler.

Parameters **to_strip** (*str, optional*) –

Returns New QueryCompiler containing the result of execution of the "rstrip" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.rstrip` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

str_slice(*start=None*, *stop=None*, *step=None*)

Apply "slice" function to each string value in QueryCompiler.

Parameters

- **start** (*int, optional*) –
- **stop** (*int, optional*) –
- **step** (*int, optional*) –

Returns New QueryCompiler containing the result of execution of the "slice" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.slice` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_slice_replace(*start=None*, *stop=None*, *repl=None*)
Apply "slice_replace" function to each string value in QueryCompiler.

Parameters

- **start** (*int, optional*) –
- **stop** (*int, optional*) –
- **repl** (*str or callable, optional*) –

Returns New QueryCompiler containing the result of execution of the "slice_replace" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.slice_replace` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_split(*pat=None*, *n=- 1*, *expand=False*)
Apply "split" function to each string value in QueryCompiler.

Parameters

- **pat** (*str, optional*) –
- **n** (*int, default: -1*) –
- **expand** (*bool, default: False*) –

Returns New QueryCompiler containing the result of execution of the "split" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.split` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_startswith(*pat*, *na=nan*)

Apply "startswith" function to each string value in QueryCompiler.

Parameters

- **pat** (*str*) –
- **na** (*object, default: np.NaN*) –

Returns New QueryCompiler containing the result of execution of the "startswith" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.startswith` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_strip(*to_strip=None*)

Apply "strip" function to each string value in QueryCompiler.

Parameters **to_strip** (*str, optional*) –

Returns New QueryCompiler containing the result of execution of the "strip" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.strip` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_swapcase()

Apply "swapcase" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "swapcase" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.swapcase` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_title()`
Apply "title" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "title" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.title` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_translate(table)`
Apply "translate" function to each string value in QueryCompiler.

Parameters **table** (*dict*) –

Returns New QueryCompiler containing the result of execution of the "translate" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.translate` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

`str_upper()`
Apply "upper" function to each string value in QueryCompiler.

Returns New QueryCompiler containing the result of execution of the "upper" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.upper` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_wrap(*width*, ***kwargs*)
Apply "wrap" function to each string value in QueryCompiler.

Parameters

- **width** (*int*) –
- ****kwargs** (*dict*) –

Returns New QueryCompiler containing the result of execution of the "wrap" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.wrap` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

str_zfill(*width*)
Apply "zfill" function to each string value in QueryCompiler.

Parameters **width** (*int*) –

Returns New QueryCompiler containing the result of execution of the "zfill" function against each string element.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.str.zfill` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

sub(*other*, ***kwargs*)
Perform element-wise substraction (`self - other`).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (BaseQueryCompiler, *scalar or array-like*) – Other operand of the binary operation.

- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.
- **axis** (*{0, 1}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.
- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

sum(***kwargs*)

Get the sum for each column or row.

Parameters

- **axis** (*{0, 1}*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the sum for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.sum` for more information about parameters and output format.

sum_min_count(***kwargs*)

Get the sum for each column or row.

Parameters

- **axis** (*{0, 1}*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the sum for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.sum` for more information about parameters and output format.

to_datetime(**args*, ***kwargs*)
Convert columns of the QueryCompiler to the datetime dtype.

Parameters

- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns QueryCompiler with all columns converted to datetime dtype.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.to_datetime` for more information about parameters and output format.

to_numeric(**args*, ***kwargs*)
Convert underlying data to numeric dtype.

Parameters

- **errors** (*{"ignore", "raise", "coerce"}*) –
- **downcast** (*{"integer", "signed", "unsigned", "float", None}*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with converted to numeric values.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.to_numeric` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

to_numpy(***kwargs*)
Convert underlying query compilers data to NumPy array.

Parameters

- **dtype** (*dtype*) – The dtype of the resulted array.
- **copy** (*bool*) – Whether to ensure that the returned value is not a view on another array.
- **na_value** (*object*) – The value to replace missing values with.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns The QueryCompiler converted to NumPy array.

Return type np.ndarray

abstract to_pandas()
Convert underlying query compilers data to pandas.DataFrame.

Returns The QueryCompiler converted to pandas.

Return type pandas.DataFrame

transpose(*args, **kwargs)
Transpose this QueryCompiler.

Parameters

- **copy** (*bool*) – Whether to copy the data after transposing.
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Transposed new QueryCompiler.

Return type *BaseQueryCompiler*

truediv(*other*, **kwargs)
Perform element-wise division (self / other).

If axes are not equal, perform frames alignment first.

Parameters

- **other** (BaseQueryCompiler, *scalar or array-like*) – Other operand of the binary operation.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that is passed from a high-level API.
- **level** (*int or label*) – In case of MultiIndex match index values on the passed level.
- **axis** (*{{0, 1}}*) – Axis to match indices along for 1D *other* (list or QueryCompiler that represents Series). 0 is for index, when 1 is for columns.
- **fill_value** (*float or None*) – Value to fill missing elements during frame alignment.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Result of binary operation.

Return type *BaseQueryCompiler*

unique(**kwargs)
Get unique values of *self*.

Parameters ****kwargs** (*dict*) – Serves compatibility purpose. Does not affect the result.

Returns New QueryCompiler with unique values.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.unique` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

unstack(*level*, *fill_value*)
Pivot a level of the (necessarily hierarchical) index labels.

Parameters

- **level** (*int or label*) –
- **fill_value** (*scalar or dict*) –

Returns

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.unstack` for more information about parameters and output format.

var(***kwargs*)
Get the variance for each column or row.

Parameters

- **axis** (*{{0, 1}}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –
- **skipna** (*bool, default: True*) –
- **ddof** (*int*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the variance for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.var` for more information about parameters and output format.

view(*index=None*, *columns=None*)
Mask QueryCompiler with passed keys.

Parameters

- **index** (*list-like of ints, optional*) – Positional indices of rows to grab.

- **columns** (*list-like of ints, optional*) – Positional indices of columns to grab.

Returns New masked QueryCompiler.

Return type *BaseQueryCompiler*

where(*cond*, *other*, ***kwargs*)

Update values of *self* using values from *other* at positions where *cond* is False.

Parameters

- **cond** (BaseQueryCompiler) – Boolean mask. True - keep the self value, False - replace by *other* value.
- **other** (BaseQueryCompiler *or pandas.Series*) – Object to grab replacement values from.
- **axis** (*{0, 1}*) – Axis to align frames along if axes of self, *cond* and *other* are not equal. 0 is for index, when 1 is for columns.
- **level** (*int or label, optional*) – Level of MultiIndex to align frames along if axes of self, *cond* and *other* are not equal. Currently *level* parameter is not implemented, so only None value is acceptable.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler with updated data.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.where` for more information about parameters and output format.

window_mean(*window_args*, **args*, ***kwargs*)

Create window of the specified type and compute mean for each window.

Parameters

- **window_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing mean for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the mean for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.mean` for more information about parameters and output format.

window_std(*window_args*, *ddof=1*, **args*, ***kwargs*)
Create window of the specified type and compute standart deviation for each window.

Parameters

- **window_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- **ddof** (*int, default: 1*) –
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing standart deviation for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the standart deviation for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.std` for more information about parameters and output format.

window_sum(*window_args*, **args*, ***kwargs*)
Create window of the specified type and compute sum for each window.

Parameters

- **window_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing sum for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the sum for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.sum` for more information about parameters and output format.

window_var(*window_args*, *ddof=1*, **args*, ***kwargs*)
Create window of the specified type and compute variance for each window.

Parameters

- **window_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- **ddof** (*int, default: 1*) –
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing variance for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the variance for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.var` for more information about parameters and output format.

write_items(*row_numeric_index*, *col_numeric_index*, *broadcasted_items*)
Update QueryCompiler elements at the specified positions by passed values.

In contrast to `setitem` this method allows to do 2D assignments.

Parameters

- **row_numeric_index** (*list of ints*) – Row positions to write value.
- **col_numeric_index** (*list of ints*) – Column positions to write value.
- **broadcasted_items** (*2D-array*) – Values to write. Have to be same size as defined by *row_numeric_index* and *col_numeric_index*.

Returns New QueryCompiler with updated values.

Return type *BaseQueryCompiler*

Pandas storage format

PandasQueryCompiler

PandasQueryCompiler is responsible for compiling a set of known predefined functions and pairing those with dataframe algebra operators in the *PandasDataframe*, specifically for dataframes backed by `pandas.DataFrame` objects.

Each *PandasQueryCompiler* contains an instance of *PandasDataframe* which it queries to get the result.

PandasQueryCompiler supports methods built by the algebra module. If you want to add an implementation for a query compiler method, visit the algebra module documentation to see whether the new operation fits one of the existing function templates and can be easily implemented with them.

Public API

PandasQueryCompiler implements common query compilers API defined by the *BaseQueryCompiler*. Some functionalities are inherited from the base class, in the following section only overridden methods are presented.

class modin.core.storage_formats.pandas.query_compiler.**PandasQueryCompiler**(*modin_frame*)
Query compiler for the pandas storage format.

This class translates common query compiler API into the DataFrame Algebra queries, that is supposed to be executed by *PandasDataframe*.

Parameters **modin_frame** (PandasDataframe) – Modin Frame to query with the compiled queries.

abs(**args*, ***kwargs*)
Execute Map function against passed query compiler.

add(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

add_prefix(*prefix*, *axis=1*)
Add string prefix to the index labels along specified axis.

Parameters

- **prefix** (*str*) – The string to add before each label.
- **axis** (*{0, 1}, default: 1*) – Axis to add prefix along. 0 is for index and 1 is for columns.

Returns New query compiler with updated labels.

Return type *BaseQueryCompiler*

add_suffix(*suffix*, *axis=1*)
Add string suffix to the index labels along specified axis.

Parameters

- **suffix** (*str*) – The string to add after each label.

- **axis** (*{0, 1}, default: 1*) – Axis to add suffix along. 0 is for index and 1 is for columns.

Returns New query compiler with updated labels.

Return type *BaseQueryCompiler*

all(**args*, ***kwargs*)
Execute MapReduce function against passed query compiler.

any(**args*, ***kwargs*)
Execute MapReduce function against passed query compiler.

apply(*func*, *axis*, **args*, ***kwargs*)
Apply passed function across given axis.

Parameters

- **func** (*callable(pandas.Series) -> scalar, str, list or dict of such*) – The function to apply to each column or row.
- **axis** (*{0, 1}*) – Target axis to apply the function along. 0 is for index, 1 is for columns.
- ***args** (*iterable*) – Positional arguments to pass to *func*.
- ****kwargs** (*dict*) – Keyword arguments to pass to *func*.

Returns

QueryCompiler that contains the results of execution and is built by the following rules:

- Labels of specified axis are the passed functions names.
- Labels of the opposite axis are preserved.
- Each element is the result of execution of *func* against corresponding row/column.

Return type *BaseQueryCompiler*

applymap(**args*, ***kwargs*)
Execute Map function against passed query compiler.

astype(*col_dtypes*, ***kwargs*)
Convert columns dtypes to given dtypes.

Parameters

- **col_dtypes** (*dict*) – Map for column names and new dtypes.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with updated dtypes.

Return type *BaseQueryCompiler*

cat_codes()
Convert underlying categories data into its codes.

Returns New QueryCompiler containing the integer codes of the underlying categories.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.cat.codes` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

clip(*lower*, *upper*, ***kwargs*)
Trim values at input threshold.

Parameters

- **lower** (*float or list-like*) –
- **upper** (*float or list-like*) –
- **axis** (*{0, 1}*) –
- **inplace** (*{False}*) – This parameter serves the compatibility purpose. Always has to be False.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler with values limited by the specified thresholds.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.clip` for more information about parameters and output format.

columnarize()
Transpose this QueryCompiler if it has a single row but multiple columns.

This method should be called for QueryCompilers representing a Series object, i.e. `self.is_series_like()` should be True.

Returns Transposed new QueryCompiler or self.

Return type *BaseQueryCompiler*

combine(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

combine_first(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

compare(*other*, ***kwargs*)
Compare data of two QueryCompilers and highlight the difference.

Parameters

- **other** (BaseQueryCompiler) – Query compiler to compare with. Have to be the same shape and the same labeling as *self*.
- **align_axis** (*{0, 1}*) –
- **keep_shape** (*bool*) –
- **keep_equal** (*bool*) –

Returns New QueryCompiler containing the differences between *self* and passed query compiler.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.compare` for more information about parameters and output format.

concat(*axis*, *other*, ***kwargs*)
Concatenate *self* with passed query compilers along specified axis.

Parameters

- **axis** (*{0, 1}*) – Axis to concatenate along. 0 is for index and 1 is for columns.
- **other** (BaseQueryCompiler *or list of such*) – Objects to concatenate with *self*.
- **join** (*{'outer', 'inner', 'right', 'left'}, default: 'outer'*) – Type of join that will be used if indices on the other axis are different. (note: if specified, has to be passed as `join=value`).

- **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. (note: if specified, has to be passed as `ignore_index=value`).
- **sort** (*bool, default: False*) – Whether or not to sort non-concatenation axis. (note: if specified, has to be passed as `sort=value`).
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Concatenated objects.

Return type *BaseQueryCompiler*

conj(**args*, ***kwargs*)
Execute Map function against passed query compiler.

copy()
Make a copy of this object.

Returns Copy of self.

Return type *BaseQueryCompiler*

Notes

For copy, we don't want a situation where we modify the metadata of the copies if we end up modifying something here. We copy all of the metadata to prevent that.

corr(*method='pearson'*, *min_periods=1*)
Compute pairwise correlation of columns, excluding NA/null values.

Parameters

- **method** (*{'pearson', 'kendall', 'spearman'} or callable(pandas.Series, pandas.Series) -> pandas.Series*) – Correlation method.
- **min_periods** (*int*) – Minimum number of observations required per pair of columns to have a valid result. If fewer than *min_periods* non-NA values are present the result will be NA.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Correlation matrix.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.corr` for more information about parameters and output format.

count(**args*, ***kwargs*)
Execute MapReduce function against passed query compiler.

cov(*min_periods=None*)
Compute pairwise covariance of columns, excluding NA/null values.

Parameters

- **min_periods** (*int*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Covariance matrix.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.cov` for more information about parameters and output format.

cummax(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

cummin(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

cumprod(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

cumsum(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

default_to_pandas(*pandas_op*, **args*, ***kwargs*)
Do fallback to pandas for the passed function.

Parameters

- **pandas_op** (*callable(pandas.DataFrame) -> object*) – Function to apply to the casted to pandas frame.
- ***args** (*iterable*) – Positional arguments to pass to *pandas_op*.
- ****kwargs** (*dict*) – Key-value arguments to pass to *pandas_op*.

Returns The result of the *pandas_op*, converted back to `BaseQueryCompiler`.

Return type *BaseQueryCompiler*

describe(***kwargs*)
Generate descriptive statistics.

Parameters

- **percentiles** (*list-like*) –
- **include** (*"all" or list of dtypes, optional*) –
- **exclude** (*list of dtypes, optional*) –
- **datetime_is_numeric** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler object containing the descriptive statistics of the underlying data.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.describe` for more information about parameters and output format.

df_update(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

diff(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

dot(*other*, *squeeze_self=None*, *squeeze_other=None*)
Compute the matrix multiplication of *self* and *other*.

Parameters

- **other** (BaseQueryCompiler *or NumPy array*) – The other query compiler or NumPy array to matrix multiply with *self*.
- **squeeze_self** (*boolean*) – If *self* is a one-column query compiler, indicates whether it represents Series object.
- **squeeze_other** (*boolean*) – If *other* is a one-column query compiler, indicates whether it represents Series object.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns A new query compiler that contains result of the matrix multiply.

Return type *BaseQueryCompiler*

drop(*index=None*, *columns=None*)
Drop specified rows or columns.

Parameters

- **index** (*list of labels, optional*) – Labels of rows to drop.
- **columns** (*list of labels, optional*) – Labels of columns to drop.

Returns New QueryCompiler with removed data.

Return type *BaseQueryCompiler*

dropna(***kwargs*)
Remove missing values.

Parameters

- **axis** (*{0, 1}*) –
- **how** (*{"any", "all"}*) –
- **thresh** (*int, optional*) –
- **subset** (*list of labels*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with null values dropped along given axis.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.dropna` for more information about parameters and output format.

dt_ceil(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_date(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_day(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_day_name(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_dayofweek(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_dayofyear(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_days(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_days_in_month(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_daysinmonth(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_end_time(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_floor(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_freq()
Get the time frequency of the underlying time-series data.

Returns QueryCompiler containing a single value, the frequency of the data.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.freq` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

dt_hour(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_is_leap_year(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_is_month_end(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_is_month_start(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_is_quarter_end(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_is_quarter_start(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_is_year_end(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_is_year_start(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_microsecond(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_microseconds(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_minute(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_month(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_month_name(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_nanosecond(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_nanoseconds(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_normalize(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_quarter(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_qyear(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_round(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_second(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_seconds(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_start_time(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_strftime(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_time(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_timetz(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_to_period(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_to_pydatetime(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_to_pytimedelta(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_to_timestamp(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_total_seconds(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_tz()
Get the time-zone of the underlying time-series data.

Returns QueryCompiler containing a single value, time-zone of the data.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.dt.tz` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

dt_tz_convert(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_tz_localize(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_week(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_weekday(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_weekofyear(**args*, ***kwargs*)
Execute Map function against passed query compiler.

dt_year(**args*, ***kwargs*)
Execute Map function against passed query compiler.

property dtypes
Get columns dtypes.

Returns Series with dtypes of each column.

Return type pandas.Series

eq(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

eval(*expr*, ***kwargs*)
Evaluate string expression on QueryCompiler columns.

Parameters

- **expr** (*str*) –
- ****kwargs** (*dict*) –

Returns QueryCompiler containing the result of evaluation.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.eval` for more information about parameters and output format.

explode(*column*)
Explode the given columns.

Parameters **column** (*Union[Hashable, Sequence[Hashable]]*) – The columns to explode.

Returns QueryCompiler that contains the results of execution. For each row in the input QueryCompiler, if the selected columns each contain M items, there will be M rows created by exploding the columns.

Return type *BaseQueryCompiler*

fillna(***kwargs*)
Replace NaN values using provided method.

Parameters

- **value** (*scalar or dict*) –
- **method** (*{"backfill", "bfill", "pad", "ffill", None}*) –
- **axis** (*{0, 1}*) –
- **inplace** (*{False}*) – This parameter serves the compatibility purpose. Always has to be False.
- **limit** (*int, optional*) –
- **downcast** (*dict, optional*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with all null values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.fillna` for more information about parameters and output format.

finalize()
Finalize constructing the dataframe calling all deferred functions which were used to build it.

first_valid_index()
Return index label of first non-NaN/NULL value.

Returns

Return type scalar

floordiv(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

free()
Trigger a cleanup of this object.

classmethod from_arrow(*at*, *data_cls*)
Build QueryCompiler from Arrow Table.

Parameters

- **at** (*Arrow Table*) – The Arrow Table to convert from.
- **data_cls** (*type*) – *PandasDataframe* class (or its descendant) to convert to.

Returns QueryCompiler containing data from the pandas DataFrame.

Return type *BaseQueryCompiler*

classmethod from_pandas(*df*, *data_cls*)
Build QueryCompiler from pandas DataFrame.

Parameters

- **df** (*pandas.DataFrame*) – The pandas DataFrame to convert from.
- **data_cls** (*type*) – *PandasDataframe* class (or its descendant) to convert to.

Returns QueryCompiler containing data from the pandas DataFrame.

Return type *BaseQueryCompiler*

ge(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

get_dummies(*columns*, ***kwargs*)
Convert categorical variables to dummy variables for certain columns.

Parameters

- **columns** (*label or list of such*) – Columns to convert.
- **prefix** (*str or list of such*) –
- **prefix_sep** (*str*) –
- **dummy_na** (*bool*) –
- **drop_first** (*bool*) –
- **dtype** (*dtype*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with categorical variables converted to dummy.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.get_dummies` for more information about parameters and output format.

`getitem_array`(*key*)
Mask QueryCompiler with *key*.

Parameters **key** (`BaseQueryCompiler, np.ndarray or list of column labels`) – Boolean mask represented by QueryCompiler or `np.ndarray` of the same shape as *self*, or enumerable of columns to pick.

Returns New masked QueryCompiler.

Return type *BaseQueryCompiler*

`getitem_column_array`(*key*, *numeric=False*)
Get column data for target labels.

Parameters

- **key** (`list-like`) – Target labels by which to retrieve data.
- **numeric** (`bool, default: False`) – Whether or not the key passed in represents the numeric index or the named index.

Returns New QueryCompiler that contains specified columns.

Return type *BaseQueryCompiler*

`getitem_row_array`(*key*)
Get row data for target indices.

Parameters **key** (`list-like`) – Numeric indices of the rows to pick.

Returns New QueryCompiler that contains specified rows.

Return type *BaseQueryCompiler*

`groupby_agg`(*by*, *is_multi_by*, *axis*, *agg_func*, *agg_args*, *agg_kwargs*, *groupby_kwargs*, *drop=False*)
Group QueryCompiler data and apply passed aggregation function.

Parameters

- **by** (`BaseQueryCompiler, column or index label, Grouper or list of such`) – Object that determine groups.
- **is_multi_by** (`bool`) – If *by* is a QueryCompiler or list of such indicates whether it's grouping on multiple columns/rows.
- **axis** (`{0, 1}`) – Axis to group and apply aggregation function along. 0 is for index, when 1 is for columns.
- **agg_func** (`dict or callable(DataFrameGroupBy) -> DataFrame`) – Function to apply to the GroupBy object.
- **agg_args** (`dict`) – Positional arguments to pass to the *agg_func*.
- **agg_kwargs** (`dict`) – Key arguments to pass to the *agg_func*.
- **groupby_kwargs** (`dict`) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.

- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns QueryCompiler containing the result of groupby aggregation.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.GroupBy.aggregate` for more information about parameters and output format.

groupby_all(***kwargs*)

Group QueryCompiler data and check whether all elements are True for every group.

Parameters

- **by** (`BaseQueryCompiler`, *column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the boolean of whether all elements are True for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.all` for more information about parameters and output format.

groupby_any(***kwargs*)

Group QueryCompiler data and check whether any element is True for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the boolean of whether there is any element which is True for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.any` for more information about parameters and output format.

groupby_count(***kwargs*)

Group QueryCompiler data and count non-null values for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the number of non-null values for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.count` for more information about parameters and output format.

groupby_max(***kwargs*)
Group QueryCompiler data and get the maximum value for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the maximum value for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.max` for more information about parameters and output format.

groupby_min(***kwargs*)
Group QueryCompiler data and get the minimum value for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the minimum value for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.min` for more information about parameters and output format.

groupby_prod(***kwargs*)
Group QueryCompiler data and compute product for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the product for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.prod` for more information about parameters and output format.

groupby_size(*by*, *axis*, *groupby_args*, *map_args*, *reduce_args*, *numeric_only*, *drop*)
Group QueryCompiler data and get the number of elements for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the number of elements for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.size` for more information about parameters and output format.

groupby_sum(***kwargs*)

Group QueryCompiler data and compute sum for every group.

Parameters

- **by** (*BaseQueryCompiler, column or index label, Grouper or list of such*) – Object that determine groups.
- **axis** (*{0, 1}*) – Axis to group and apply reduction function along. 0 is for index, when 1 is for columns.
- **groupby_args** (*dict*) – GroupBy parameters as expected by `modin.pandas.DataFrame.groupby` signature.
- **map_args** (*dict*) – Keyword arguments to pass to the reduction function. If GroupBy is implemented via MapReduce approach, this argument is passed at the map phase only.
- **reduce_args** (*dict, optional*) – If GroupBy is implemented with MapReduce approach, specifies arguments to pass to the reduction function at the reduce phase, has no effect otherwise.
- **numeric_only** (*bool, default: True*) – Whether or not to drop non-numeric columns before executing GroupBy.
- **drop** (*bool, default: False*) – If *by* is a QueryCompiler indicates whether or not by-data came from the *self*.

Returns

- *BaseQueryCompiler* – QueryCompiler containing the result of groupby reduction built by the following rules:
  - Labels on the opposit of specified axis are preserved.
  - If groupby_args["as_index"] is True then labels on the specified axis are the group names, otherwise labels would be default: 0, 1 … n.
  - If groupby_args["as_index"] is False, then first N columns/rows of the frame contain group names, where N is the columns/rows to group on.
  - Each element of QueryCompiler is the sum for the corresponding group and column/row.
- .. *warning* – *map_args* and *reduce_args* parameters are deprecated. They're leaked here from `PandasQueryCompiler.groupby_*`, pandas storage format implements groupby via MapReduce approach, but for other storage formats these parameters make no sense, and so they'll be removed in the future.

Notes

Please refer to `modin.pandas.GroupBy.sum` for more information about parameters and output format.

gt(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

idxmax(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

idxmin(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

insert(*loc*, *column*, *value*)
Insert new column.

Parameters

- **loc** (*int*) – Insertion position.
- **column** (*label*) – Label of the new column.
- **value** (*One-column BaseQueryCompiler, 1D array or scalar*) – Data to fill new column with.

Returns QueryCompiler with new column inserted.

Return type *BaseQueryCompiler*

invert(**args*, ***kwargs*)
Execute Map function against passed query compiler.

is_monotonic_decreasing()
Return boolean if values in the object are monotonicly decreasing.

Returns

Return type bool

is_monotonic_increasing()
Return boolean if values in the object are monotonicly increasing.

Returns

Return type bool

is_series_like()
Check whether this QueryCompiler can represent `modin.pandas.Series` object.

Returns Return True if QueryCompiler has a single column or row, False otherwise.

Return type bool

isin(**args*, ***kwargs*)
Execute Map function against passed query compiler.

isna(**args*, ***kwargs*)
Execute Map function against passed query compiler.

join(*right*, ***kwargs*)
Join columns of another QueryCompiler.

Parameters

- **right** (`BaseQueryCompiler`) – QueryCompiler of the right frame to join with.
- **on** (*label or list of such*) –
- **how** (*{"left", "right", "outer", "inner"}*) –
- **lsuffix** (*str*) –
- **rsuffix** (*str*) –
- **sort** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler that contains result of the join.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.join` for more information about parameters and output format.

kurt(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

last_valid_index()
Return index label of last non-NaN/NULL value.

Returns

Return type scalar

le(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.

- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

lt(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

mad(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

max(*axis*, ***kwargs*)
Get the maximum value for each column or row.

Parameters

- **axis** (*{{0, 1}}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –
- **skipna** (*bool, default: True*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the maximum value for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.max` for more information about parameters and output format.

mean(*axis*, ***kwargs*)
Get the mean value for each column or row.

Parameters

- **axis** (*{{0, 1}}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –
- **skipna** (*bool, default: True*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the mean value for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.mean` for more information about parameters and output format.

median(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

melt(*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name='value'*, *col_level=None*, *ignore_index=True*)
Unpivot QueryCompiler data from wide to long format.

Parameters

- **id_vars** (*list of labels, optional*) –
- **value_vars** (*list of labels, optional*) –
- **var_name** (*label*) –
- **value_name** (*label*) –
- **col_level** (*int or label*) –
- **ignore_index** (*bool*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with unpivoted data.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.melt` for more information about parameters and output format.

memory_usage(**args*, ***kwargs*)
Execute MapReduce function against passed query compiler.

merge(*right*, ***kwargs*)
Merge QueryCompiler objects using a database-style join.

Parameters

- **right** (*BaseQueryCompiler*) – QueryCompiler of the right frame to merge with.
- **how** (*{"left", "right", "outer", "inner", "cross"}*) –
- **on** (*label or list of such*) –
- **left_on** (*label or list of such*) –
- **right_on** (*label or list of such*) –
- **left_index** (*bool*) –
- **right_index** (*bool*) –
- **sort** (*bool*) –
- **suffixes** (*list-like*) –
- **copy** (*bool*) –
- **indicator** (*bool or str*) –
- **validate** (*str*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler that contains result of the merge.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.merge` for more information about parameters and output format.

min(*axis*, ***kwargs*)
Get the minimum value for each column or row.

Parameters

- **axis** (*{{0, 1}}*) –
- **level** (*None, default: None*) – Serves the compatibility purpose. Always has to be None.
- **numeric_only** (*bool, optional*) –
- **skipna** (*bool, default: True*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the minimum value for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.min` for more information about parameters and output format.

mod(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

mode(***kwargs*)
Get the modes for every column or row.

Parameters

- **axis** (*{0, 1}*) –
- **numeric_only** (*bool*) –
- **dropna** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler with modes calculated alogn given axis.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.mode` for more information about parameters and output format.

mul(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.

- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

ne(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

negative(**args*, ***kwargs*)
Execute Map function against passed query compiler.

nlargest(**args*, ***kwargs*)
Return the first *n* rows ordered by *columns* in descending order.

Parameters

- **n** (*int, default: 5*) –
- **columns** (*list of labels, optional*) – Column labels to order by. (note: this parameter can be omitted only for a single-column query compilers representing Series object, otherwise *columns* has to be specified).
- **keep** (*{"first", "last", "all"}, default: "first"*) –

Returns

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.nlargest` for more information about parameters and output format.

notna(**args*, ***kwargs*)
Execute Map function against passed query compiler.

nsmallest(**args*, ***kwargs*)
Return the first *n* rows ordered by *columns* in ascending order.

Parameters

- **n** (*int, default: 5*) –
- **columns** (*list of labels, optional*) – Column labels to order by. (note: this parameter can be omitted only for a single-column query compilers representing Series object, otherwise *columns* has to be specified).
- **keep** (*{"first", "last", "all"}, default: "first"*) –

Returns

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.nsmallest` for more information about parameters and output format.

nunique(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

pivot(*index*, *columns*, *values*)
Produce pivot table based on column values.

Parameters

- **index** (*label or list of such, pandas.Index, optional*) –
- **columns** (*label or list of such*) –
- **values** (*label or list of such, optional*) –

Returns New QueryCompiler containing pivot table.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.pivot` for more information about parameters and output format.

pivot_table(*index*, *values*, *columns*, *aggfunc*, *fill_value*, *margins*, *dropna*, *margins_name*, *observed*, *sort*)
Create a spreadsheet-style pivot table from underlying data.

Parameters

- **index** (*label, pandas.Grouper, array or list of such*) –
- **values** (*label, optional*) –
- **columns** (*column, pandas.Grouper, array or list of such*) –

- **aggfunc** (*callable(pandas.Series) -> scalar, dict of list of such*) –
- **fill_value** (*scalar, optional*) –
- **margins** (*bool*) –
- **dropna** (*bool*) –
- **margins_name** (*str*) –
- **observed** (*bool*) –
- **sort** (*bool*) –

Returns

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.pivot_table` for more information about parameters and output format.

pow(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

prod(**args*, ***kwargs*)
Execute MapReduce function against passed query compiler.

prod_min_count(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

quantile_for_list_of_values(***kwargs*)
Get the value at the given quantile for each column or row.

Parameters

- **q** (*list-like*) –
- **axis** (*{0, 1}*) –
- **numeric_only** (*bool*) –
- **interpolation** (*{"linear", "lower", "higher", "midpoint", "nearest"}*) –

- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler with index labels of the specified axis, where each row contains the value at the given quantile for the corresponding row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.quantile` for more information about parameters and output format.

quantile_for_single_value(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

query(*expr*, ***kwargs*)
Query columns of the QueryCompiler with a boolean expression.

Parameters

- **expr** (*str*) –
- ****kwargs** (*dict*) –

Returns New QueryCompiler containing the rows where the boolean expression is satisfied.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.query` for more information about parameters and output format.

rank(***kwargs*)
Compute numerical rank along the specified axis.

By default, equal values are assigned a rank that is the average of the ranks of those values, this behaviour can be changed via *method* parameter.

Parameters

- **axis** (*{0, 1}*) –
- **method** (*{"average", "min", "max", "first", "dense"}*) –
- **numeric_only** (*bool*) –
- **na_option** (*{"keep", "top", "bottom"}*) –
- **ascending** (*bool*) –
- **pct** (*bool*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler of the same shape as *self*, where each element is the numerical rank of the corresponding value along row or column.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.rank` for more information about parameters and output format.

reindex(*axis*, *labels*, ***kwargs*)
Align QueryCompiler data with a new index along specified axis.

Parameters

- **axis** (*{0, 1}*) – Axis to align labels along. 0 is for index, 1 is for columns.
- **labels** (*list-like*) – Index-labels to align with.
- **method** (*{None, "backfill"/"bfill", "pad"/"ffill", "nearest"}*) – Method to use for filling holes in reindexed frame.
- **fill_value** (*scalar*) – Value to use for missing values in the resulted frame.
- **limit** (*int*) –
- **tolerance** (*int*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler with aligned axis.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.reindex` for more information about parameters and output format.

replace(**args*, ***kwargs*)
Execute Map function against passed query compiler.

resample_agg_df(*resample_args*, *func*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and apply passed aggregation function for each group over the specified axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **func** (*str, dict, callable(pandas.Series) -> scalar, or list of such*) –
- ***args** (*iterable*) – Positional arguments to pass to the aggregation function.
- ****kwargs** (*dict*) – Keyword arguments to pass to the aggregation function.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are a MultiIndex, where first level contains preserved labels of this axis and the second level is the function names.

- Each element of QueryCompiler is the result of corresponding function for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.agg` for more information about parameters and output format.

resample_agg_ser(*resample_args*, *func*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and apply passed aggregation function in a one-column query compiler for each group over the specified axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **func** (*str, dict, callable(pandas.Series) -> scalar, or list of such*) –
- ***args** (*iterable*) – Positional arguments to pass to the aggregation function.
- ****kwargs** (*dict*) – Keyword arguments to pass to the aggregation function.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are a MultiIndex, where first level contains preserved labels of this axis and the second level is the function names.
- Each element of QueryCompiler is the result of corresponding function for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.agg` for more information about parameters and output format.

> **Warning:** This method duplicates logic of `resample_agg_df` and will be removed soon.

resample_app_df(*resample_args*, *func*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and apply passed aggregation function for each group over the specified axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.

- **func** (*str, dict, callable(pandas.Series) -> scalar, or list of such*) –
- ***args** (*iterable*) – Positional arguments to pass to the aggregation function.
- ****kwargs** (*dict*) – Keyword arguments to pass to the aggregation function.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are a MultiIndex, where first level contains preserved labels of this axis and the second level is the function names.
- Each element of QueryCompiler is the result of corresponding function for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.apply` for more information about parameters and output format.

Warning: This method duplicates logic of `resample_agg_df` and will be removed soon.

resample_app_ser(*resample_args*, *func*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and apply passed aggregation function in a one-column query compiler for each group over the specified axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **func** (*str, dict, callable(pandas.Series) -> scalar, or list of such*) –
- ***args** (*iterable*) – Positional arguments to pass to the aggregation function.
- ****kwargs** (*dict*) – Keyword arguments to pass to the aggregation function.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are a MultiIndex, where first level contains preserved labels of this axis and the second level is the function names.
- Each element of QueryCompiler is the result of corresponding function for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.apply` for more information about parameters and output format.

Warning: This method duplicates logic of `resample_agg_df` and will be removed soon.

resample_asfreq(*resample_args*, *fill_value*)
Resample time-series data and get the values at the new frequency.

Group data into intervals by time-series row/column with a specified frequency and get values at the new frequency.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **fill_value** (*scalar*) –

Returns New QueryCompiler containing values at the specified frequency.

Return type *BaseQueryCompiler*

resample_backfill(*resample_args*, *limit*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using back-fill method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **limit** (*int*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.backfill` for more information about parameters and output format.

resample_bfill(*resample_args*, *limit*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using back-fill method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.

- **limit** (*int*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.bfill` for more information about parameters and output format.

resample_count(*resample_args*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute number of non-NA values for each group.

Parameters **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the number of non-NA values for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.count` for more information about parameters and output format.

resample_ffill(*resample_args*, *limit*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using forward-fill method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **limit** (*int*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.ffill` for more information about parameters and output format.

resample_fillna(*resample_args*, *method*, *limit*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using specified method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **method** (*str*) –
- **limit** (*int*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.fillna` for more information about parameters and output format.

resample_first(*resample_args*, *_method*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute first element for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the first element for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.first` for more information about parameters and output format.

resample_get_group(*resample_args*, *name*, *obj*)
Resample time-series data and get the specified group.

Group data into intervals by time-series row/column with a specified frequency and get the values of the specified group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **name** (*object*) –
- **obj** (*modin.pandas.DataFrame, optional*) –

Returns New QueryCompiler containing the values from the specified group.

Return type *BaseQueryCompiler*

resample_interpolate(*resample_args*, *method*, *axis*, *limit*, *inplace*, *limit_direction*, *limit_area*, *downcast*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using specified interpolation method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **method** (*str*) –
- **axis** (*{0, 1}*) –
- **limit** (*int*) –
- **inplace** (*{False}*) – This parameter serves the compatibility purpose. Always has to be False.
- **limit_direction** (*{"forward", "backward", "both"}*) –
- **limit_area** (*{None, "inside", "outside"}*) –
- **downcast** (*str, optional*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.interpolate` for more information about parameters and output format.

resample_last(*resample_args*, *_method*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute last element for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the last element for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.last` for more information about parameters and output format.

resample_max(*resample_args*, *_method*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute maximum value for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)

- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the maximum value for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.max` for more information about parameters and output format.

resample_mean(*resample_args*, *_method*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute mean value for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the mean value for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.mean` for more information about parameters and output format.

resample_median(*resample_args*, *_method*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute median value for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the median value for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.median` for more information about parameters and output format.

resample_min(*resample_args*, *_method*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute minimum value for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the minimum value for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.min` for more information about parameters and output format.

resample_nearest(*resample_args*, *limit*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using 'nearest' method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **limit** (*int*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.nearest` for more information about parameters and output format.

resample_nunique(*resample_args*, *_method*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute number of unique values for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the number of unique values for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.nunique` for more information about parameters and output format.

resample_ohlc_df(*resample_args*, *_method*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute open, high, low and close values for each group over the specified axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Positional arguments to pass to the aggregation function.
- ****kwargs** (*dict*) – Keyword arguments to pass to the aggregation function.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are a MultiIndex, where first level contains preserved labels of this axis and the second level is the labels of columns containing computed values.
- Each element of QueryCompiler is the result of corresponding function for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.ohlc` for more information about parameters and output format.

resample_ohlc_ser(*resample_args*, *_method*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute open, high, low and close values for each group over the specified axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- ***args** (*iterable*) – Positional arguments to pass to the aggregation function.
- ****kwargs** (*dict*) – Keyword arguments to pass to the aggregation function.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are a MultiIndex, where first level contains preserved labels of this axis and the second level is the labels of columns containing computed values.
- Each element of QueryCompiler is the result of corresponding function for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.ohlc` for more information about parameters and output format.

resample_pad(*resample_args*, *limit*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and fill missing values in each group independently using 'pad' method.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **limit** (*int*) –

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- QueryCompiler contains unsampled data with missing values filled.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.pad` for more information about parameters and output format.

resample_pipe(*resample_args*, *func*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency, build equivalent `pandas.Resampler` object and apply passed function to it.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **func** (*callable(pandas.Resampler) -> object or tuple(callable, str)*) –
- ***args** (*iterable*) – Positional arguments to pass to function.
- ****kwargs** (*dict*) – Keyword arguments to pass to function.

Returns New QueryCompiler containing the result of passed function.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Resampler.pipe` for more information about parameters and output format.

resample_prod(*resample_args*, *_method*, *min_count*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute product for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- **min_count** (*int*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the product for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.prod` for more information about parameters and output format.

resample_quantile(*resample_args*, *q*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute quantile for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **q** (*float*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the quantile for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.quantile` for more information about parameters and output format.

resample_sem(*resample_args*, *_method*, **args*, ***kwargs*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute standart error of the mean for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **ddof** (*int, default: 1*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the standart error of the mean for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.sem` for more information about parameters and output format.

resample_size(*resample_args*)
Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute number of elements in a group for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the number of elements in a group for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.size` for more information about parameters and output format.

resample_std(*resample_args*, *ddof*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute standart deviation for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **ddof** (*int*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the standart deviation for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.std` for more information about parameters and output format.

resample_sum(*resample_args*, *_method*, *min_count*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute sum for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **_method** (*str*) –
- **min_count** (*int*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the sum for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.sum` for more information about parameters and output format.

resample_transform(*resample_args*, *arg*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and call passed function on each group. In contrast to `resample_app_df` apply function to the whole group, instead of a single axis.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **arg** (*callable(pandas.DataFrame) -> pandas.Series*) –
- ***args** (*iterable*) – Positional arguments to pass to function.
- ****kwargs** (*dict*) – Keyword arguments to pass to function.

Returns New QueryCompiler containing the result of passed function.

Return type *BaseQueryCompiler*

resample_var(*resample_args*, *ddof*, **args*, ***kwargs*)

Resample time-series data and apply aggregation on it.

Group data into intervals by time-series row/column with a specified frequency and compute variance for each group.

Parameters

- **resample_args** (*list*) – Resample parameters as expected by `modin.pandas.DataFrame.resample` signature.
- **ddof** (*int*) –
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.

- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns

New QueryCompiler containing the result of resample aggregation built by the following rules:

- Labels on the specified axis are the group names (time-stamps)
- Labels on the opposit of specified axis are preserved.
- Each element of QueryCompiler is the variance for the corresponding group and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.Resampler.var` for more information about parameters and output format.

reset_index(***kwargs*)
Reset the index, or a level of it.

Parameters

- **drop** (*bool*) – Whether to drop the reset index or insert it at the beginning of the frame.
- **level** (*int or label, optional*) – Level to remove from index. Removes all levels by default.
- **col_level** (*int or label*) – If the columns have multiple levels, determines which level the labels are inserted into.
- **col_fill** (*label*) – If the columns have multiple levels, determines how the other levels are named.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler with reset index.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.reset_index` for more information about parameters and output format.

rfloordiv(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.

- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

rmod(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

rolling_aggregate(*rolling_args*, *func*, **args*, ***kwargs*)
Create rolling window and apply specified functions for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- **func** (*str, dict, callable(pandas.Series) -> scalar, or list of such*) –
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing the result of passed functions for each window, built by the following rulles:

- Labels on the specified axis are preserved.
- Labels on the opposit of specified axis are MultiIndex, where first level contains preserved labels of this axis and the second level has the function names.
- Each element of QueryCompiler is the result of corresponding function for the corresponding window and column/row.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.aggregate` for more information about parameters and output format.

rolling_apply(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

rolling_corr(*rolling_args*, *other*, *pairwise*, **args*, ***kwargs*)
Create rolling window and compute correlation for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- **other** (*modin.pandas.Series, modin.pandas.DataFrame, list-like, optional*) –
- **pairwise** (*bool, optional*) –
- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing correlation for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the correlation for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.corr` for more information about parameters and output format.

rolling_count(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

rolling_cov(*rolling_args*, *other*, *pairwise*, *ddof*, ***kwargs*)
Create rolling window and compute covariance for each window.

Parameters

- **rolling_args** (*list*) – Rolling windows arguments with the same signature as `modin.pandas.DataFrame.rolling`.
- **other** (*modin.pandas.Series, modin.pandas.DataFrame, list-like, optional*) –
- **pairwise** (*bool, optional*) –
- **ddof** (*int, default: 1*) –
- ****kwargs** (*dict*) –

Returns

New QueryCompiler containing covariance for each window, built by the following rulles:

- Output QueryCompiler has the same shape and axes labels as the source.
- Each element is the covariance for the corresponding window.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Rolling.cov` for more information about parameters and output format.

rolling_kurt(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

rolling_max(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

rolling_mean(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

rolling_median(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

rolling_min(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

rolling_quantile(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

rolling_skew(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

rolling_std(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

rolling_sum(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

rolling_var(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

round(**args*, ***kwargs*)
Execute Map function against passed query compiler.

rpow(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

rsub(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

rtruediv(*other*, *broadcast=False*, **args*, ***kwargs*)

Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

searchsorted(***kwargs*)

Find positions in a sorted *self* where *value* should be inserted to maintain order.

Parameters

- **value** (*list-like*) –
- **side** (*{"left", "right"}*) –
- **sorter** (*list-like, optional*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns One-column QueryCompiler which contains indices to insert.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.searchsorted` for more information about parameters and output format.

> **Warning:** This method is supported only by one-column query compilers.

sem(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

series_update(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

series_view(**args*, ***kwargs*)
Execute Map function against passed query compiler.

set_index_from_columns(*keys: List[Hashable]*, *drop: bool = True*, *append: bool = False*)
Create new row labels from a list of columns.

Parameters

- **keys** (*list of hashable*) – The list of column names that will become the new index.
- **drop** (*bool, default: True*) – Whether or not to drop the columns provided in the *keys* argument.
- **append** (*bool, default: True*) – Whether or not to add the columns in *keys* as new levels appended to the existing index.

Returns A new QueryCompiler with updated index.

Return type *BaseQueryCompiler*

setitem(*axis*, *key*, *value*)
Set the row/column defined by *key* to the *value* provided.

Parameters

- **axis** (*{0, 1}*) – Axis to set *value* along. 0 means set row, 1 means set column.
- **key** (*label*) – Row/column label to set *value* in.

- **value** (*BaseQueryCompiler, list-like or scalar*) – Define new row/column value.

Returns New QueryCompiler with updated *key* value.

Return type *BaseQueryCompiler*

skew(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

sort_columns_by_row_values(*rows*, *ascending=True*, ***kwargs*)
Reorder the columns based on the lexicographic order of the given rows.

Parameters

- **rows** (*label or list of labels*) – The row or rows to sort by.
- **ascending** (*bool, default: True*) – Sort in ascending order (True) or descending order (False).
- **kind** (*{"quicksort", "mergesort", "heapsort"}*) –
- **na_position** (*{"first", "last"}*) –
- **ignore_index** (*bool*) –
- **key** (*callable(pandas.Index) -> pandas.Index, optional*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler that contains result of the sort.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.sort_values` for more information about parameters and output format.

sort_index(***kwargs*)
Sort data by index or column labels.

Parameters

- **axis** (*{0, 1}*) –
- **level** (*int, label or list of such*) –
- **ascending** (*bool*) –
- **inplace** (*bool*) –
- **kind** (*{"quicksort", "mergesort", "heapsort"}*) –
- **na_position** (*{"first", "last"}*) –
- **sort_remaining** (*bool*) –
- **ignore_index** (*bool*) –
- **key** (*callable(pandas.Index) -> pandas.Index, optional*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler containing the data sorted by columns or indices.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.sort_index` for more information about parameters and output format.

sort_rows_by_column_values(*columns*, *ascending=True*, ***kwargs*)
Reorder the rows based on the lexicographic order of the given columns.

Parameters

- **columns** (*label or list of labels*) – The column or columns to sort by.
- **ascending** (*bool, default: True*) – Sort in ascending order (True) or descending order (False).
- **kind** (*{"quicksort", "mergesort", "heapsort"}*) –
- **na_position** (*{"first", "last"}*) –
- **ignore_index** (*bool*) –
- **key** (*callable(pandas.Index) -> pandas.Index, optional*) –
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns New QueryCompiler that contains result of the sort.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.sort_values` for more information about parameters and output format.

stack(*level*, *dropna*)
Stack the prescribed level(s) from columns to index.

Parameters

- **level** (*int or label*) –
- **dropna** (*bool*) –

Returns

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.stack` for more information about parameters and output format.

std(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

str___getitem__(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_capitalize(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_center(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_contains(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_count(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_endswith(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_find(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_findall(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_get(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_index(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_isalnum(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_isalpha(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_isdecimal(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_isdigit(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_islower(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_isnumeric(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_isspace(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_istitle(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_isupper(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_join(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_len(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_ljust(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_lower(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_lstrip(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_match(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_normalize(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_pad(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_partition(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_repeat(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_replace(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_rfind(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_rindex(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_rjust(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_rpartition(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_rsplit(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_rstrip(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_slice(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_slice_replace(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_split(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_startswith(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_strip(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_swapcase(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_title(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_translate(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_upper(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_wrap(**args*, ***kwargs*)
Execute Map function against passed query compiler.

str_zfill(**args*, ***kwargs*)
Execute Map function against passed query compiler.

sub(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

sum(**args*, ***kwargs*)
Execute MapReduce function against passed query compiler.

sum_min_count(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

to_datetime(**args*, ***kwargs*)
Convert columns of the QueryCompiler to the datetime dtype.

Parameters

- ***args** (*iterable*) –
- ****kwargs** (*dict*) –

Returns QueryCompiler with all columns converted to datetime dtype.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.to_datetime` for more information about parameters and output format.

to_numeric(**args*, ***kwargs*)
Execute Map function against passed query compiler.

to_numpy(***kwargs*)
Convert underlying query compilers data to NumPy array.

Parameters

- **dtype** (*dtype*) – The dtype of the resulted array.
- **copy** (*bool*) – Whether to ensure that the returned value is not a view on another array.
- **na_value** (*object*) – The value to replace missing values with.

- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns The QueryCompiler converted to NumPy array.

Return type np.ndarray

to_pandas()
Convert underlying query compilers data to `pandas.DataFrame`.

Returns The QueryCompiler converted to pandas.

Return type pandas.DataFrame

transpose(**args*, ***kwargs*)
Transpose this QueryCompiler.

Parameters

- **copy** (*bool*) – Whether to copy the data after transposing.
- ***args** (*iterable*) – Serves the compatibility purpose. Does not affect the result.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns Transposed new QueryCompiler.

Return type *BaseQueryCompiler*

truediv(*other*, *broadcast=False*, **args*, ***kwargs*)
Apply binary *func* to passed operands.

Parameters

- **query_compiler** (*QueryCompiler*) – Left operand of *func*.
- **other** (*QueryCompiler, list-like object or scalar*) – Right operand of *func*.
- **broadcast** (*bool, default: False*) – If *other* is a one-column query compiler, indicates whether it is a Series or not. Frames and Series have to be processed differently, however we can't distinguish them at the query compiler level, so this parameter is a hint that passed from a high level API.
- ***args** (*args,*) – Arguments that will be passed to *func*.
- ****kwargs** (*kwargs,*) – Arguments that will be passed to *func*.

Returns Result of binary function.

Return type QueryCompiler

unique()
Get unique values of *self*.

Parameters ****kwargs** (*dict*) – Serves compatibility purpose. Does not affect the result.

Returns New QueryCompiler with unique values.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.Series.unique` for more information about parameters and output format.

Warning: This method is supported only by one-column query compilers.

unstack(*level*, *fill_value*)
Pivot a level of the (necessarily hierarchical) index labels.

Parameters

- **level** (*int or label*) –
- **fill_value** (*scalar or dict*) –

Returns

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.unstack` for more information about parameters and output format.

var(**args*, ***kwargs*)
Execute Reduction function against passed query compiler.

view(*index=None*, *columns=None*)
Mask QueryCompiler with passed keys.

Parameters

- **index** (*list-like of ints, optional*) – Positional indices of rows to grab.
- **columns** (*list-like of ints, optional*) – Positional indices of columns to grab.

Returns New masked QueryCompiler.

Return type *BaseQueryCompiler*

where(*cond*, *other*, ***kwargs*)
Update values of *self* using values from *other* at positions where *cond* is False.

Parameters

- **cond** (BaseQueryCompiler) – Boolean mask. True - keep the self value, False - replace by *other* value.
- **other** (BaseQueryCompiler *or pandas.Series*) – Object to grab replacement values from.
- **axis** (*{0, 1}*) – Axis to align frames along if axes of self, *cond* and *other* are not equal. 0 is for index, when 1 is for columns.
- **level** (*int or label, optional*) – Level of MultiIndex to align frames along if axes of self, *cond* and *other* are not equal. Currently *level* parameter is not implemented, so only None value is acceptable.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns QueryCompiler with updated data.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.where` for more information about parameters and output format.

window_mean(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

window_std(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

window_sum(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

window_var(**args*, ***kwargs*)
Execute Fold function against passed query compiler.

write_items(*row_numeric_index*, *col_numeric_index*, *broadcasted_items*)
Update QueryCompiler elements at the specified positions by passed values.

In contrast to `setitem` this method allows to do 2D assignments.

Parameters

- **row_numeric_index** (*list of ints*) – Row positions to write value.
- **col_numeric_index** (*list of ints*) – Column positions to write value.
- **broadcasted_items** (*2D-array*) – Values to write. Have to be same size as defined by *row_numeric_index* and *col_numeric_index*.

Returns New QueryCompiler with updated values.

Return type *BaseQueryCompiler*

Pandas Parsers Module Description

High-Level Module Overview

This module houses parser classes (classes that are used for data parsing on the workers) and util functions for handling parsing results. `PandasParser` is base class for parser classes with pandas storage format, that contains methods common for all child classes. Other module classes implement `parse` function that performs parsing of specific format data basing on the chunk information computed in the `modin.core.io` module. After chunk data parsing is completed, resulting `DataFrame`-s will be splitted into smaller `DataFrame`-s according to `num_splits` parameter, data type and number or rows/columns in the parsed chunk, and then these frames and some additional metadata will be returned.

Note: If you are interested in the data parsing mechanism implementation details, please refer to the source code documentation.

High-Level Module Overview

This module houses submodules which are responsible for communication between the query compiler level and execution implementation level for pandas storage format:

- *Query compiler* is responsible for compiling efficient queries for *PandasDataframe*.
- *Parsers* are responsible for parsing data on workers during IO operations.

PyArrow storage format

PyarrowQueryCompiler

PyarrowQueryCompiler is responsible for compiling efficient Dataframe algebra queries for the PyarrowOnRayDataframe, the frames which are backed by `pyarrow.Table` objects.

Each *PyarrowQueryCompiler* contains an instance of `PyarrowOnRayDataframe` which it queries to get the result.

Public API

PyarrowQueryCompiler implements common query compilers API defined by the *BaseQueryCompiler*. Most functionalities are inherited from *PandasQueryCompiler*, in the following section only overridden methods are presented.

class modin.core.storage_formats.pyarrow.query_compiler.**PyarrowQueryCompiler**(*modin_frame*)

Bases: *modin.core.storage_formats.pandas.query_compiler.PandasQueryCompiler*

Query compiler for the PyArrow storage format.

This class translates common query compiler API into the DataFrame Algebra queries, that is supposed to be executed by `PyarrowOnRayDataframe`.

Parameters **modin_frame** (*PyarrowOnRayDataframe*) – Modin Frame to query with the compiled queries.

property dtypes

Get columns dtypes.

Returns Series with dtypes of each column.

Return type pandas.Series

query(*expr*, ***kwargs*)

Query columns of the QueryCompiler with a boolean expression.

Parameters

- **expr** (*str*) –
- ****kwargs** (*dict*) –

Returns New QueryCompiler containing the rows where the boolean expression is satisfied.

Return type *BaseQueryCompiler*

Notes

Please refer to `modin.pandas.DataFrame.query` for more information about parameters and output format.

to_numpy(***kwargs*)
Convert underlying query compilers data to NumPy array.

Parameters

- **dtype** (*dtype*) – The dtype of the resulted array.
- **copy** (*bool*) – Whether to ensure that the returned value is not a view on another array.
- **na_value** (*object*) – The value to replace missing values with.
- ****kwargs** (*dict*) – Serves the compatibility purpose. Does not affect the result.

Returns The QueryCompiler converted to NumPy array.

Return type np.ndarray

to_pandas()
Convert underlying query compilers data to `pandas.DataFrame`.

Returns The QueryCompiler converted to pandas.

Return type pandas.DataFrame

PyArrow Parsers Module Description

This module houses parser classes that are responsible for data parsing on the workers for the PyArrow storage format. Parsers for PyArrow storage formats follow an interface of *pandas format parsers*: parser class of every file format implements `parse` method, which parses the specified part of the file and builds PyArrow tables from the parsed data, based on the specified chunk size and number of splits. The resulted PyArrow tables will be used as a partitions payload in the `PyarrowOnRayDataframe`.

Public API

Module houses Modin parser classes, that are used for data parsing on the workers.

class modin.core.storage_formats.pyarrow.parsers.**PyarrowCSVParser**
Class for handling CSV files on the workers using PyArrow storage format.

parse(*fname*, *num_splits*, *start*, *end*, *header*, ***kwargs*)
Parse CSV file into PyArrow tables.

Parameters

- **fname** (*str*) – Name of the CSV file to parse.
- **num_splits** (*int*) – Number of partitions to split the resulted PyArrow table into.
- **start** (*int*) – Position in the specified file to start parsing from.
- **end** (*int*) – Position in the specified file to end parsing at.
- **header** (*str*) – Header line that will be interpret as the first line of the parsed CSV file.
- ****kwargs** (*kwargs*) – Serves the compatibility purpose. Does not affect the result.

Returns

List with splitted parse results and it's metadata:

- First *num_split* elements are PyArrow tables, representing the corresponding chunk.
- Next element is the number of rows in the parsed table.
- Last element is the pandas Series, containing the data-types for each column of the parsed table.

Return type list

In general, PyArrow storage formats follow the flow of the pandas ones: query compiler contains an instance of Modin Dataframe, which is internally split into partitions. The main difference is that partitions contain PyArrow tables, instead of `pandas.DataFrame`-s like with *pandas storage format*. To learn more about this approach please visit PyArrowOnRay execution section.

High-Level Module Overview

This module houses submodules which are responsible for communication between the query compiler level and execution implementation level for PyArrow storage format:

- *Query compiler* is responsible for compiling efficient queries for PyarrowOnRayDataframe.
- *Parsers* are responsible for parsing data on workers during IO operations.

Note: Currently the only one available PyArrow storage format factory is `PyarrowOnRay` which works in experimental mode only.

Modin supports several execution backends (storage format + execution engine). Calling any DataFrame API function will end up in some execution-specific method. The query compiler is a bridge between pandas DataFrame API and the actual Core Modin Dataframe implementation for the corresponding execution.

Each storage format has its own Query Compiler class that implements the most optimal query routing for the selected format.

Query compilers of all storage formats implement a common API, which is used by the high-level Modin DataFrame to support dataframe queries. The role of the query compiler is to translate its API into a pairing of known user-defined functions and dataframe algebra operators. Each query compiler instance contains a Core Modin Dataframe of the selected execution implementation and queries it with the compiled queries to get the result. The query compiler object is immutable, so the result of every method is a new query compiler.

The query compilers API is defined by the `BaseQueryCompiler` class and may resemble the pandas API, however, they're not equal. The query compilers API is significantly reduced in comparison with pandas, since many corner cases or even the whole methods can be handled at the API layer with the existing API.

The query compiler is the level where Modin stops distinguishing DataFrame and Series (or column) objects. A Series is represented by a *1xN* query compiler, where the Series name is the column label. If Series is unnamed, then the label is `"__reduced__"`. The high-level DataFrame API layer interprets a one-column query compiler as Series or DataFrame depending on the operation context.

Note: Although we're declaring that there is no difference between DataFrame and Series at the query compiler, you still may find methods like `method_ser` and `method_df` which are implemented differently because they're emulating

either Series or DataFrame logic, or you may find parameters, which indicates whether this one-column query compiler is representing Series or not. All of these are hacks, and we're working on getting rid of them.

High-level module overview

This module houses submodules of all of the stable storage formats:

- *Base module* contains an abstract query compiler class which defines common API.
- *Pandas module* contains query compiler and text parsers for pandas storage format.
- cuDF module contains query compiler and text parsers for cuDF storage format.
- *Pyarrow module* contains query compiler and text parsers for Pyarrow storage format.

You can find more in the experimental section.

PandasOnPython Dataframe implementation

This page describes implementation of *Modin PandasDataframe Objects* specific for *PandasOnPython* execution. Since Python engine doesn't allow computation parallelization, operations on partitions are performed sequentially. The absence of parallelization doesn't give any perfomance speed-up, so `PandasOnPython` is used for testing purposes only.

- *PandasOnPythonDataframe*
- *PandasOnPythonDataframePartition*
- *PandasOnPythonDataframeAxisPartition*
- *PandasOnPythonDataframePartitionManager*

PandasOnPythonDataframe

The class is specific implementation of `PandasDataframe` for *Python* execution engine. It serves as an intermediate level between `PandasQueryCompiler` and `PandasOnPythonDataframePartitionManager`.

Public API

class modin.core.execution.python.implementations.pandas_on_python.dataframe.dataframe.**PandasOnPythonDa**

Class for dataframes with pandas storage format and Python engine.

`PandasOnPythonDataframe` doesn't implement any specific interfaces, all functionality is inherited from the `PandasDataframe` class.

Parameters

- **partitions** (*np.ndarray*) – A 2D NumPy array of partitions.

- **index** (*sequence*) – The index for the dataframe. Converted to a `pandas.Index`.
- **columns** (*sequence*) – The columns object for the dataframe. Converted to a `pandas.Index`.
- **row_lengths** (*list, optional*) – The length of each partition in the rows. The "height" of each of the block partitions. Is computed if not provided.
- **column_widths** (*list, optional*) – The width of each partition in the columns. The "width" of each of the block partitions. Is computed if not provided.
- **dtypes** (*pandas.Series, optional*) – The data types for the dataframe columns.

PandasOnPythonDataframePartition

The class is specific implementation of *PandasDataframePartition*, providing the API to perform operations on a block partition using Python as the execution engine.

In addition to wrapping a `pandas.DataFrame`, the class also holds the following metadata:

- `length` - length of `pandas.DataFrame` wrapped
- `width` - width of `pandas.DataFrame` wrapped

An operation on a block partition can be performed in two modes:

- immediately via *apply()* - in this case accumulated call queue and new function will be executed immediately.
- lazily via *add_to_apply_calls()* - in this case function will be added to the call queue and no computations will be done at the moment.

Public API

class modin.core.execution.python.implementations.pandas_on_python.partitioning.partition.**PandasOnPytho**

Partition class with interface for pandas storage format and Python engine.

Class holds the data and metadata for a single partition and implements methods of parent abstract class `PandasDataframePartition`.

Parameters

- **data** (*pandas.DataFrame*) – `pandas.DataFrame` that should be wrapped with this class.
- **length** (*int, optional*) – Length of *data* (number of rows in the input dataframe).
- **width** (*int, optional*) – Width of *data* (number of columns in the input dataframe).
- **call_queue** (*list, optional*) – Call queue of the partition (list with entities that should be called before partition materialization).

Notes

Objects of this class are treated as immutable by partition manager subclasses. There is no logic for updating in-place.

add_to_apply_calls(*func*, **args*, ***kwargs*)
Add a function to the call queue.

Parameters

- **func** (*callable*) – Function to be added to the call queue.
- ***args** (*iterable*) – Additional positional arguments to be passed in *func*.
- ****kwargs** (*dict*) – Additional keyword arguments to be passed in *func*.

Returns New PandasOnPythonDataframePartition object with extended call queue.

Return type *PandasOnPythonDataframePartition*

apply(*func*, **args*, ***kwargs*)
Apply a function to the object wrapped by this partition.

Parameters

- **func** (*callable*) – Function to apply.
- ***args** (*iterable*) – Additional positional arguments to be passed in *func*.
- ****kwargs** (*dict*) – Additional keyword arguments to be passed in *func*.

Returns New PandasOnPythonDataframePartition object.

Return type *PandasOnPythonDataframePartition*

drain_call_queue()
Execute all operations stored in the call queue on the object wrapped by this partition.

classmethod empty()
Create a new partition that wraps an empty pandas DataFrame.

Returns New PandasOnPythonDataframePartition object wrapping empty pandas DataFrame.

Return type *PandasOnPythonDataframePartition*

get()
Flush the *call_queue* and return copy of the data.

Returns Copy of DataFrame that was wrapped by this partition.

Return type pandas.DataFrame

Notes

Since this object is a simple wrapper, just return the copy of data.

length()
Get the length of the object wrapped by this partition.

Returns The length of the object.

Return type int

classmethod preprocess_func(*func*)
Preprocess a function before an apply call.

Parameters **func** (*callable*) – Function to preprocess.

Returns An object that can be accepted by `apply`.

Return type callable

Notes

No special preprocessing action is required, so unmodified *func* will be returned.

classmethod put(*obj*)
Create partition containing *obj*.

Parameters **obj** (*pandas.DataFrame*) – DataFrame to be put into the new partition.

Returns New `PandasOnPythonDataframePartition` object.

Return type *PandasOnPythonDataframePartition*

to_numpy(***kwargs*)
Return NumPy array representation of `pandas.DataFrame` stored in this partition.

Parameters ****kwargs** (*dict*) – Keyword arguments to pass into *pandas.DataFrame.to_numpy* function.

Returns

Return type np.ndarray

to_pandas()
Return copy of the `pandas.Dataframe` stored in this partition.

Returns

Return type pandas.DataFrame

Notes

Equivalent to `get` method for this class.

wait()
Wait for completion of computations on the object wrapped by the partition.

Internally will be done by flushing the call queue.

width()
Get the width of the object wrapped by the partition.

Returns The width of the object.

Return type int

PandasOnPythonDataframeAxisPartition

The class is specific implementation of *PandasDataframeAxisPartition*, providing the API to perform operations on an axis partition, using Python as the execution engine. The axis partition is made up of list of block partitions that are stored in this class.

Public API

class modin.core.execution.python.implementations.pandas_on_python.partitioning.axis_partition.**PandasOnl**
Class defines axis partition interface with pandas storage format and Python engine.

Inherits functionality from `PandasDataframeAxisPartition` class.

Parameters **list_of_blocks** (*list*) – List with partition objects to create common axis partition from.

PandasOnPythonFrameColumnPartition

Public API

class modin.core.execution.python.implementations.pandas_on_python.partitioning.axis_partition.**PandasOnl**
The column partition implementation for pandas storage format and Python engine.

All of the implementation for this class is in the `PandasOnPythonDataframeAxisPartition` parent class, and this class defines the axis to perform the computation over.

Parameters **list_of_blocks** (*list*) – List with partition objects to create common axis partition from.

PandasOnPythonFrameRowPartition

Public API

class modin.core.execution.python.implementations.pandas_on_python.partitioning.axis_partition.**PandasOnl**
The row partition implementation for pandas storage format and Python engine.

All of the implementation for this class is in the `PandasOnPythonDataframeAxisPartition` parent class, and this class defines the axis to perform the computation over.

Parameters **list_of_blocks** (*list*) – List with partition objects to create common axis partition from.

PandasOnPythonDataframePartition

The class is specific implementation of *PandasDataframePartitionManager* using Python as the execution engine. This class is responsible for partitions manipulation and applying a funcion to block/row/column partitions.

Public API

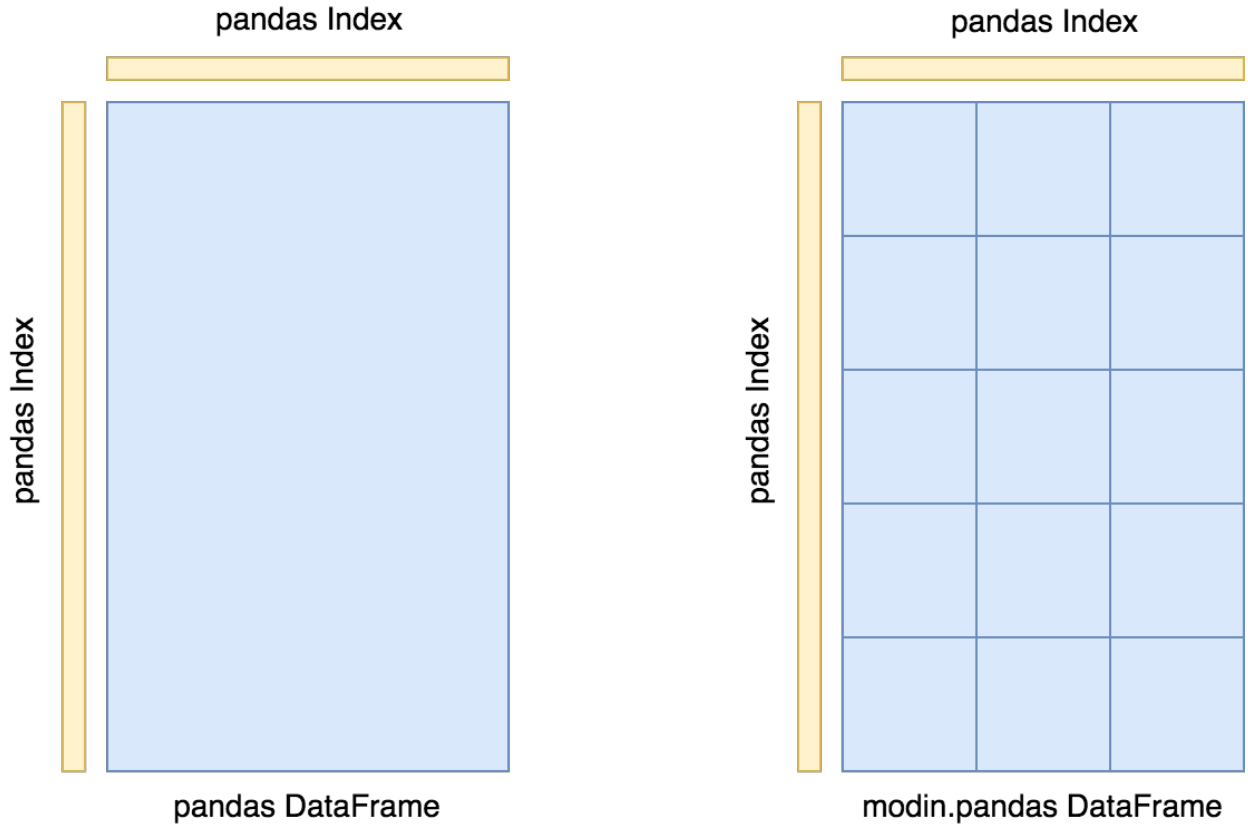
class modin.core.execution.python.implementations.pandas_on_python.partitioning.partition_manager.**PandasOnPythonDataframePartitionManager**

Class for managing partitions with pandas storage format and Python engine.

Inherits all functionality from `PandasDataframePartitionManager` base class.

3.20.5 DataFrame Partitioning

The Modin DataFrame architecture follows in the footsteps of modern architectures for database and high performance matrix systems. We chose a partitioning schema that partitions along both columns and rows because it gives Modin flexibility and scalability in both the number of columns and the number of rows supported. The following figure illustrates this concept.



Currently, each partition's memory format is a pandas DataFrame. In the future, we will support additional in-memory formats, namely Arrow tables.

Index

We currently use the `pandas.Index` object for both indexing columns and rows. In the future, we will implement a distributed, pandas-compatible Index object in order remove this scaling limitation from the system. It does not start to become a problem until you are operating on more than 10's of billions of columns or rows, so most workloads will not be affected by this scalability limit. **Important note**: If you are using the default index (`pandas.RangeIndex`) there is a fixed memory overhead (~200 bytes) and there will be no scalability issues with the index.

API

The API is the outer-most layer that faces users. The majority of our current effort is spent implementing the components of the pandas API. We have implemented a toy example for a sqlite API as a proof of concept, but this isn't ready for usage/testing. There are also plans to expose the Modin DataFrame API as a reduced API set that encompasses the entire pandas/dataframe API. See experimental features for more information.

BasePandasDataset

The class implements functionality that is common to Modin's pandas API for both `DataFrame` and `Series` classes.

Public API

class modin.pandas.base.**BasePandasDataset**

Implement most of the common code that exists in DataFrame/Series.

Since both objects share the same underlying representation, and the algorithms are the same, we use this object to define the general behavior of those objects and then use those objects to define the output type.

Notes

See pandas API documentation for pandas.DataFrame for more.

abs()

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

Returns Series/DataFrame containing the absolute value of each element.

Return type abs

See also:

numpy.absolute Calculate the absolute value element-wise.

Notes

See pandas API documentation for pandas.DataFrame.abs for more. For `complex` inputs, `1.2 + 1j`, the absolute value is $\sqrt{a^2 + b^2}$.

Examples

Absolute numeric values in a Series.

```
>>> s = pd.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 Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
     a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
3    7   40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
     a    b    c
1    5   20   50
0    4   10  100
2    6   30  -30
3    7   40  -50
```

add(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Get Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *radd*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.add for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
```

(continues on next page)

(continued from previous page)

```
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

agg(*func=None*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

- **func** (*function, str, list or dict*) – Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

  Accepted combinations are:

  – function

  - string function name
  - list of functions and/or function names, e.g. `[np.sum, 'mean']`
  - dict of axis labels -> functions, function names or list of such.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.
- ***args** – Positional arguments to pass to *func*.
- ****kwargs** – Keyword arguments to pass to *func*.

Returns

- *scalar, Series or DataFrame* – The return can be:
  - scalar : when Series.agg is called with single function
  - Series : when DataFrame.agg is called with a single function
  - DataFrame : when DataFrame.agg is called with several functions

  Return scalar, Series or DataFrame.
- *The aggregation operations are always performed over an axis, either the*
- *index (default) or the column axis. This behavior is different from*
- *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*,
- *var*), where the default is to compute the aggregation of the flattened
- array, e.g., `numpy.mean(arr_2d)` as opposed to
- `numpy.mean(arr_2d, axis=0)`.
- *agg* is an alias for *aggregate*. Use the alias.

See also:

DataFrame.apply Perform any type of operations.

DataFrame.transform Perform transformation type operations.

core.groupby.GroupBy Perform operations over groups.

core.resample.Resampler Perform operations over resampled bins.

core.window.Rolling Perform operations over rolling window.

core.window.Expanding Perform operations over expanding window.

core.window.ExponentialMovingWindow Perform operation over exponential weighted window.

Notes

See pandas API documentation for pandas.DataFrame.aggregate for more. *agg* is an alias for *aggregate*. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See gotchas.udf-mutation for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
sum  12.0  NaN
min   1.0  2.0
max   NaN  8.0
```

Aggregate different functions over the columns and rename the index of the resulting DataFrame.

```
>>> df.agg(x=('A', max), y=('B', 'min'), z=('C', np.mean))
     A    B    C
x  7.0  NaN  NaN
y  NaN  2.0  NaN
z  NaN  NaN  6.0
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

aggregate(*func=None*, *axis=0*, **args*, ***kwargs*)

Aggregate using one or more operations over the specified axis.

Parameters

- **func** (*function, str, list or dict*) – Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply.

  Accepted combinations are:

  - function
  - string function name
  - list of functions and/or function names, e.g. `[np.sum, 'mean']`
  - dict of axis labels -> functions, function names or list of such.

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.
- ***args** – Positional arguments to pass to *func*.
- ****kwargs** – Keyword arguments to pass to *func*.

Returns

- *scalar, Series or DataFrame* – The return can be:
  - scalar : when Series.agg is called with single function
  - Series : when DataFrame.agg is called with a single function
  - DataFrame : when DataFrame.agg is called with several functions

  Return scalar, Series or DataFrame.
- *The aggregation operations are always performed over an axis, either the*
- *index (default) or the column axis. This behavior is different from*
- *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*,
- *var*), where the default is to compute the aggregation of the flattened
- array, e.g., `numpy.mean(arr_2d)` as opposed to
- `numpy.mean(arr_2d, axis=0)`.
- *agg* is an alias for *aggregate*. Use the alias.

See also:

DataFrame.apply Perform any type of operations.

DataFrame.transform Perform transformation type operations.

core.groupby.GroupBy Perform operations over groups.

core.resample.Resampler Perform operations over resampled bins.

core.window.Rolling Perform operations over rolling window.

core.window.Expanding Perform operations over expanding window.

core.window.ExponentialMovingWindow Perform operation over exponential weighted window.

Notes

See pandas API documentation for pandas.DataFrame.aggregate for more. *agg* is an alias for *aggregate*. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See gotchas.udf-mutation for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
sum  12.0  NaN
min   1.0  2.0
max   NaN  8.0
```

Aggregate different functions over the columns and rename the index of the resulting DataFrame.

```
>>> df.agg(x=('A', max), y=('B', 'min'), z=('C', np.mean))
     A    B    C
x  7.0  NaN  NaN
y  NaN  2.0  NaN
z  NaN  NaN  6.0
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0    2.0
1    5.0
2    8.0
3    NaN
dtype: float64
```

align(*other*, *join='outer'*, *axis=None*, *level=None*, *copy=True*, *fill_value=None*, *method=None*, *limit=None*, *fill_axis=0*, *broadcast_axis=None*)

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).
- **level** (*int or level name, default None*) – Broadcast across a level, matching Index values on the passed MultiIndex level.

- **copy** (*bool, default True*) – Always returns new objects. If copy=False and no reindexing is required then original objects are returned.
- **fill_value** (*scalar, default np.NaN*) – Value to use for missing values. Defaults to NaN, but can be any "compatible" value.
- **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.
- **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.
- **fill_axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – Filling axis, method and limit.
- **broadcast_axis** (*{0 or 'index', 1 or 'columns'}, default None*) – Broadcast values along this axis, if aligning two objects of different dimensions.

Returns **(left, right)** – Aligned objects.

Return type (DataFrame, type of other)

Notes

See pandas API documentation for pandas.DataFrame.align for more.

all(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)

Return whether all elements are True, potentially over an axis.

Returns True unless there at least one element within a series or along a Dataframe axis that is False or equivalent (e.g. zero or empty).

Parameters

- **axis** (*{0 or 'index', 1 or 'columns', None}, 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.
  - 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
  - None : reduce all axes, return a scalar.
- **bool_only** (*bool, default None*) – Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.
- **skipna** (*bool, default True*) – Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be True, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- ****kwargs** (*any, default None*) – Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns If level is specified, then, DataFrame is returned; otherwise, Series is returned.

Return type *Series* or DataFrame

See also:

`Series.all` Return True if all elements are True.

`DataFrame.any` Return True if one (or more) elements are True.

Examples

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
>>> pd.Series([], dtype="float64").all()
True
>>> pd.Series([np.nan]).all()
True
>>> pd.Series([np.nan]).all(skipna=False)
True
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

Notes

See pandas API documentation for pandas.DataFrame.all for more.

any(*axis=0*, *bool_only=None*, *skipna=True*, *level=None*, ***kwargs*)
Return whether any element is True, potentially over an axis.

Returns False unless there is at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

Parameters

- **axis** (*{0 or 'index', 1 or 'columns', None}, 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.
  - 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
  - None : reduce all axes, return a scalar.
- **bool_only** (*bool, default None*) – Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.
- **skipna** (*bool, default True*) – Exclude NA/null values. If the entire row/column is NA and skipna is True, then the result will be False, as for an empty row/column. If skipna is False, then NA are treated as True, because these are not equal to zero.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- ****kwargs** (*any, default None*) – Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns If level is specified, then, DataFrame is returned; otherwise, Series is returned.

Return type *Series* or DataFrame

See also:

numpy.any Numpy version of this method.

Series.any Return whether any element is True.

Series.all Return whether all elements are True.

DataFrame.any Return whether any element is True over requested axis.

DataFrame.all Return whether all elements are True over requested axis.

Examples

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
>>> pd.Series([], dtype="float64").any()
```

(continues on next page)

(continued from previous page)

```
False
>>> pd.Series([np.nan]).any()
False
>>> pd.Series([np.nan]).any(skipna=False)
True
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
       A  B
0   True  1
1  False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
       A  B
0   True  1
1  False  0
```

```
>>> df.any(axis='columns')
0    True
1    False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

Notes

See pandas API documentation for pandas.DataFrame.any for more.

apply(*func*, *axis=0*, *broadcast=None*, *raw=False*, *reduce=None*, *result_type=None*, *convert_dtype=True*, *args=()*, ***kwds*)

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`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the *result_type* argument.

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.
- **raw** (*bool, default False*) – Determines if row or column is passed as a Series or ndarray object:
  - `False` : passes each row or column as a Series to the function.
  - `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.
- **result_type** (*{'expand', 'reduce', 'broadcast', None}, default None*) – These only act when `axis=1` (columns):
  - 'expand' : list-like results will be turned into columns.
  - 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
  - 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

  The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.
- **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 Result of applying `func` along the given axis of the DataFrame.

Return type *Series* or DataFrame

See also:

DataFrame.applymap For elementwise operations.

DataFrame.aggregate Only perform aggregating type operations.

DataFrame.transform Only perform transforming type operations.

Notes

See pandas API documentation for pandas.DataFrame.apply for more. Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See gotchas.udf-mutation for more details.

Examples

```
>>> df = pd.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)`):

```
>>> df.apply(np.sqrt)
     A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```

Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
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
```

Passing `result_type='expand'` will expand list-like results to columns of a Dataframe

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
   0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0    1    2
1    1    2
2    1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
1  1  2
2  1  2
```

asfreq(*freq*, *method=None*, *how=None*, *normalize=False*, *fill_value=None*)

Convert time series to specified frequency.

Returns the original data conformed to a new index with the specified frequency.

If the index of this DataFrame is a `PeriodIndex`, the new index is the result of transforming the original index with `PeriodIndex.asfreq` (so the original index will map one-to-one to the new index).

Otherwise, the new index will be equivalent to `pd.date_range(start, end, freq=freq)` where `start` and `end` are, respectively, the first and last entries in the original index (see `pandas.date_range()`). The values corresponding to any timesteps in the new index which were not present in the original index will be null (`NaN`), unless a method for filling such unknowns is provided (see the `method` parameter below).

The `resample()` method is more appropriate if an operation on each group of timesteps (such as an aggregate) is necessary to represent the data at the new frequency.

Parameters

- **freq** (*DateOffset or str*) – Frequency DateOffset or string.
- **method** (*{'backfill'/'bfill', 'pad'/'ffill'}, default None*) – Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):
  - 'pad' / 'ffill': propagate last valid observation forward to next valid
  - 'backfill' / 'bfill': use NEXT valid observation to fill.
- **how** (*{'start', 'end'}, default end*) – For PeriodIndex only (see PeriodIndex.asfreq).
- **normalize** (*bool, default False*) – Whether to reset output index to midnight.
- **fill_value** (*scalar, optional*) – Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

Returns DataFrame object reindexed to the specified frequency.

Return type DataFrame

See also:

reindex Conform DataFrame to new index with optional filling logic.

Notes

See pandas API documentation for pandas.DataFrame.asfreq for more. To learn more about the frequency strings, please see this link.

Examples

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s': series})
>>> df
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:01:00  NaN
2000-01-01 00:02:00  2.0
2000-01-01 00:03:00  3.0
```

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  NaN
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  NaN
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  NaN
2000-01-01 00:03:00  3.0
```

Upsample again, providing a `fill value`.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  9.0
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  9.0
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  9.0
2000-01-01 00:03:00  3.0
```

Upsample again, providing a `method`.

```
>>> df.asfreq(freq='30S', method='bfill')
                     s
2000-01-01 00:00:00  0.0
2000-01-01 00:00:30  NaN
2000-01-01 00:01:00  NaN
2000-01-01 00:01:30  2.0
2000-01-01 00:02:00  2.0
2000-01-01 00:02:30  3.0
2000-01-01 00:03:00  3.0
```

asof(*where*, *subset=None*)

Return the last row(s) without any NaNs before *where*.

The last row (for each element in *where*, if list) without any NaN is taken. In case of a `DataFrame`, the last row without NaN considering only the subset of columns (if not *None*)

If there is no good value, NaN is returned for a Series or a Series of NaN values for a DataFrame

Parameters

- **where** (*date or array-like of dates*) – Date(s) before which the last row(s) are returned.
- **subset** (str or array-like of str, default *None*) – For DataFrame, if not *None*, only use these columns to check for NaNs.

Returns

The return can be:

- scalar : when *self* is a Series and *where* is a scalar
- Series: when *self* is a Series and *where* is an array-like, or when *self* is a DataFrame and *where* is a scalar
- DataFrame : when *self* is a DataFrame and *where* is an array-like

Return scalar, Series, or DataFrame.

Return type scalar, *Series*, or DataFrame

See also:

merge_asof Perform an asof merge. Similar to left join.

Notes

See pandas API documentation for pandas.DataFrame.asof for more. Dates are assumed to be sorted. Raises if this is not the case.

Examples

A Series and a scalar *where*.

```
>>> s = pd.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
```

```
>>> s.asof(20)
2.0
```

For a sequence *where*, a Series is returned. The first value is NaN, because the first element of *where* is before the first index value.

```
>>> s.asof([5, 20])
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
```

Take all columns into consideration

```
>>> df = pd.DataFrame({'a': [10, 20, 30, 40, 50],
...                    'b': [None, None, None, None, 500]},
...                   index=pd.DatetimeIndex(['2018-02-27 09:01:00',
...                                           '2018-02-27 09:02:00',
...                                           '2018-02-27 09:03:00',
...                                           '2018-02-27 09:04:00',
...                                           '2018-02-27 09:05:00']))
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']))
                     a   b
2018-02-27 09:03:30 NaN NaN
2018-02-27 09:04:30 NaN NaN
```

Take a single column into consideration

```
>>> df.asof(pd.DatetimeIndex(['2018-02-27 09:03:30',
...                           '2018-02-27 09:04:30']),
...          subset=['a'])
                        a   b
2018-02-27 09:03:30   30.0 NaN
2018-02-27 09:04:30   40.0 NaN
```

astype(*dtype*, *copy=True*, *errors='raise'*)

Cast a pandas 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 pandas 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.
- **copy** (*bool, default True*) – Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).
- **errors** (*{'raise', 'ignore'}, default 'raise'*) – Control raising of exceptions on invalid data for provided dtype.
  - `raise` : allow exceptions to be raised
  - `ignore` : suppress exceptions. On error return original object.

Returns **casted**

Return type same type as caller

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to a numeric type.

numpy.ndarray.astype Cast a numpy array to a specified type.

Notes

See pandas API documentation for pandas.DataFrame.astype for more. .. deprecated:: 1.3.0

Using `astype` to convert from timezone-naive dtype to timezone-aware dtype is deprecated and will raise in a future version. Use `Series.dt.tz_localize()` instead.

Examples

Create a DataFrame:

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

Cast all columns to int32:

```
>>> df.astype('int32').dtypes
col1    int32
col2    int32
dtype: object
```

Cast col1 to int32 using a dictionary:

```
>>> df.astype({'col1': 'int32'}).dtypes
col1    int32
col2    int64
dtype: object
```

Create a series:

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> from pandas.api.types import CategoricalDtype
>>> cat_dtype = CategoricalDtype(
...     categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1, 2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1  # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

Create a series of dates:

```
>>> ser_date = pd.Series(pd.date_range('20200101', periods=3))
>>> ser_date
0   2020-01-01
1   2020-01-02
2   2020-01-03
dtype: datetime64[ns]
```

property at

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

Raises **KeyError** – If 'label' does not exist in DataFrame.

See also:

DataFrame.iat Access a single value for a row/column pair by integer position.

DataFrame.loc Access a group of rows and columns by label(s).

Series.at Access a single value using a label.

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

Notes

See pandas API documentation for pandas.DataFrame.at for more.

at_time(*time*, *asof=False*, *axis=None*)

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

Return type *Series* or DataFrame

Raises **TypeError** – If the index is not a `DatetimeIndex`

See also:

between_time Select values between particular times 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_at_time Get just the index locations for values at particular time of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
                     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
```

```
>>> ts.at_time('12:00')
                     A
2018-04-09 12:00:00  2
2018-04-10 12:00:00  4
```

Notes

See pandas API documentation for pandas.DataFrame.at_time for more.

backfill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna()` with `method='bfill'`.

Returns Object with missing values filled or None if `inplace=True`.

Return type Series/DataFrame or None

Notes

See pandas API documentation for pandas.DataFrame.backfill for more.

between_time(*start_time*, *end_time*, *include_start=True*, *include_end=True*, *axis=None*)
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 Data from the original object filtered to the specified dates range.

Return type *Series* or DataFrame

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

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
                     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
```

```
>>> ts.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`:

```
>>> ts.between_time('0:45', '0:15')
                     A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

Notes

See pandas API documentation for pandas.DataFrame.between_time for more.

bfill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)
Synonym for `DataFrame.fillna()` with `method='bfill'`.

Returns Object with missing values filled or None if `inplace=True`.

Return type Series/DataFrame or None

Notes

See pandas API documentation for pandas.DataFrame.backfill for more.

bool()
Return the bool of a single element Series or DataFrame.

This must be a boolean scalar value, either True or False. It will raise a ValueError if the Series or DataFrame does not have exactly 1 element, or that element is not boolean (integer values 0 and 1 will also raise an exception).

Returns The value in the Series or DataFrame.

Return type bool

See also:

Series.astype Change the data type of a Series, including to boolean.

DataFrame.astype Change the data type of a DataFrame, including to boolean.

numpy.bool_ NumPy boolean data type, used by pandas for boolean values.

Examples

The method will only work for single element objects with a boolean value:

```
>>> pd.Series([True]).bool()
True
>>> pd.Series([False]).bool()
False
```

```
>>> pd.DataFrame({'col': [True]}).bool()
True
>>> pd.DataFrame({'col': [False]}).bool()
False
```

Notes

See pandas API documentation for pandas.DataFrame.bool for more.

combine(*other*, *func*, *fill_value=None*, ***kwargs*)

Perform column-wise combine with another DataFrame.

Combines a DataFrame with *other* DataFrame using *func* to element-wise combine columns. The row and column indexes of the resulting DataFrame will be the union of the two.

Parameters

- **other** (*DataFrame*) – The DataFrame to merge column-wise.
- **func** (*function*) – Function that takes two series as inputs and return a Series or a scalar. Used to merge the two dataframes column by columns.
- **fill_value** (*scalar value, default None*) – The value to fill NaNs with prior to passing any column to the merge func.
- **overwrite** (*bool, default True*) – If True, columns in *self* that do not exist in *other* will be overwritten with NaNs.

Returns Combination of the provided DataFrames.

Return type DataFrame

See also:

DataFrame.combine_first Combine two DataFrame objects and default to non-null values in frame calling the method.

Examples

Combine using a simple function that chooses the smaller column.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> take_smaller = lambda s1, s2: s1 if s1.sum() < s2.sum() else s2
>>> df1.combine(df2, take_smaller)
   A  B
0  0  3
1  0  3
```

Example using a true element-wise combine function.

```
>>> df1 = pd.DataFrame({'A': [5, 0], 'B': [2, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, np.minimum)
   A  B
0  1  2
1  0  3
```

Using *fill_value* fills Nones prior to passing the column to the merge function.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
   A    B
0  0 -5.0
1  0  4.0
```

However, if the same element in both dataframes is None, that None is preserved

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [None, 3]})
>>> df1.combine(df2, take_smaller, fill_value=-5)
    A    B
0  0 -5.0
1  0  3.0
```

Example that demonstrates the use of *overwrite* and behavior when the axis differ between the dataframes.

```
>>> df1 = pd.DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [-10, 1], }, index=[1, 2])
>>> df1.combine(df2, take_smaller)
     A    B     C
0  NaN  NaN   NaN
1  NaN  3.0 -10.0
2  NaN  3.0   1.0
```

```
>>> df1.combine(df2, take_smaller, overwrite=False)
     A    B     C
0  0.0  NaN   NaN
1  0.0  3.0 -10.0
2  NaN  3.0   1.0
```

Demonstrating the preference of the passed in dataframe.

```
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1], }, index=[1, 2])
>>> df2.combine(df1, take_smaller)
   A    B   C
0  0.0  NaN NaN
1  0.0  3.0 NaN
2  NaN  3.0 NaN
```

```
>>> df2.combine(df1, take_smaller, overwrite=False)
     A    B   C
0  0.0  NaN NaN
1  0.0  3.0 1.0
2  NaN  3.0 1.0
```

Notes

See pandas API documentation for pandas.DataFrame.combine for more.

combine_first(*other*)

Update null elements with value in the same location in *other*.

Combine two DataFrame objects by filling null values in one DataFrame with non-null values from other DataFrame. The row and column indexes of the resulting DataFrame will be the union of the two.

Parameters **other** (*DataFrame*) – Provided DataFrame to use to fill null values.

Returns The result of combining the provided DataFrame with the other object.

Return type DataFrame

See also:

DataFrame.combine Perform series-wise operation on two DataFrames using a given function.

Examples

```
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [None, 4]})
>>> df2 = pd.DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine_first(df2)
     A    B
0  1.0  3.0
1  0.0  4.0
```

Null values still persist if the location of that null value does not exist in *other*

```
>>> df1 = pd.DataFrame({'A': [None, 0], 'B': [4, None]})
>>> df2 = pd.DataFrame({'B': [3, 3], 'C': [1, 1]}, index=[1, 2])
>>> df1.combine_first(df2)
     A    B    C
0  NaN  4.0  NaN
1  0.0  3.0  1.0
2  NaN  3.0  1.0
```

Notes

See pandas API documentation for pandas.DataFrame.combine_first for more.

convert_dtypes(*infer_objects: modin.pandas.base.BasePandasDataset.bool = True*, *convert_string: modin.pandas.base.BasePandasDataset.bool = True*, *convert_integer: modin.pandas.base.BasePandasDataset.bool = True*, *convert_boolean: modin.pandas.base.BasePandasDataset.bool = True*, *convert_floating: modin.pandas.base.BasePandasDataset.bool = True*)

Convert columns to best possible dtypes using dtypes supporting `pd.NA`.

New in version 1.0.0.

Parameters

- **infer_objects** (*bool, default True*) – Whether object dtypes should be converted to the best possible types.
- **convert_string** (*bool, default True*) – Whether object dtypes should be converted to `StringDtype()`.
- **convert_integer** (*bool, default True*) – Whether, if possible, conversion can be done to integer extension types.
- **convert_boolean** (*bool, defaults True*) – Whether object dtypes should be converted to `BooleanDtypes()`.
- **convert_floating** (*bool, defaults True*) – Whether, if possible, conversion can be done to floating extension types. If *convert_integer* is also True, preference will be give to integer dtypes if the floats can be faithfully casted to integers.

  New in version 1.2.0.

Returns Copy of input object with new dtype.

Return type *Series* or DataFrame

See also:

infer_objects Infer dtypes of objects.

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to a numeric type.

Notes

See pandas API documentation for pandas.DataFrame.convert_dtypes for more. By default, `convert_dtypes` will attempt to convert a Series (or each Series in a DataFrame) to dtypes that support `pd.NA`. By using the options `convert_string`, `convert_integer`, `convert_boolean` and `convert_boolean`, it is possible to turn off individual conversions to `StringDtype`, the integer extension types, `BooleanDtype` or floating extension types, respectively.

For object-dtyped columns, if `infer_objects` is `True`, use the inference rules as during normal Series/DataFrame construction. Then, if possible, convert to `StringDtype`, `BooleanDtype` or an appropriate integer or floating extension type, otherwise leave as `object`.

If the dtype is integer, convert to an appropriate integer extension type.

If the dtype is numeric, and consists of all integers, convert to an appropriate integer extension type. Otherwise, convert to an appropriate floating extension type.

Changed in version 1.2: Starting with pandas 1.2, this method also converts float columns to the nullable floating extension type.

In the future, as new dtypes are added that support `pd.NA`, the results of this method will change to support those new dtypes.

Examples

```
>>> df = pd.DataFrame(
...     {
...         "a": pd.Series([1, 2, 3], dtype=np.dtype("int32")),
...         "b": pd.Series(["x", "y", "z"], dtype=np.dtype("O")),
...         "c": pd.Series([True, False, np.nan], dtype=np.dtype("O")),
...         "d": pd.Series(["h", "i", np.nan], dtype=np.dtype("O")),
...         "e": pd.Series([10, np.nan, 20], dtype=np.dtype("float")),
...         "f": pd.Series([np.nan, 100.5, 200], dtype=np.dtype("float")),
...     }
... )
```

Start with a DataFrame with default dtypes.

```
>>> df
   a  b      c    d     e      f
0  1  x   True    h  10.0    NaN
1  2  y  False    i   NaN  100.5
2  3  z    NaN  NaN  20.0  200.0
```

```
>>> df.dtypes
a      int32
b     object
c     object
d     object
e    float64
f    float64
dtype: object
```

Convert the DataFrame to use best possible dtypes.

```
>>> dfn = df.convert_dtypes()
>>> dfn
   a  b      c     d     e      f
0  1  x   True     h    10   <NA>
1  2  y  False     i  <NA>  100.5
2  3  z   <NA>  <NA>    20  200.0
```

```
>>> dfn.dtypes
a      Int32
b     string
c    boolean
d     string
```

(continues on next page)

(continued from previous page)

```
e      Int64
f    Float64
dtype: object
```

Start with a Series of strings and missing data represented by `np.nan`.

```
>>> s = pd.Series(["a", "b", np.nan])
>>> s
0      a
1      b
2    NaN
dtype: object
```

Obtain a Series with dtype `StringDtype`.

```
>>> s.convert_dtypes()
0       a
1       b
2    <NA>
dtype: string
```

copy(*deep=True*)

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

Parameters **deep** (*bool, default True*) – Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

Returns **copy** – Object type matches caller.

Return type *Series* or DataFrame

Notes

See pandas API documentation for pandas.DataFrame.copy for more. When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to *copy.deepcopy* in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

Examples

```
>>> s = pd.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
```

Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

count(*axis=0*, *level=None*, *numeric_only=False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, and optionally *numpy.inf* (depending on *pandas.options.mode.use_inf_as_na*) 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.
- **level** (*int or str, optional*) – If the axis is a *MultiIndex* (hierarchical), count along a particular *level*, collapsing into a *DataFrame*. A *str* specifies the level name.
- **numeric_only** (*bool, default False*) – Include only *float*, *int* or *boolean* data.

Returns For each column/row the number of non-NA/null entries. If *level* is specified returns a *DataFrame*.

Return type *Series* or DataFrame

See also:

Series.count Number of non-NA elements in a Series.

DataFrame.value_counts Count unique combinations of columns.

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 = pd.DataFrame({"Person":
...                    ["John", "Myla", "Lewis", "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]})
>>> 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
```

Counts for each **row**:

```
>>> df.count(axis='columns')
0    3
1    2
2    3
3    3
4    3
dtype: int64
```

Notes

See pandas API documentation for pandas.DataFrame.count for more.

cummax(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

Parameters

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – The index or the name of the axis. 0 is equivalent to None or 'index'.
- **skipna** (*bool, default True*) – Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- ***args** – Additional keywords have no effect but might be accepted for compatibility with NumPy.
- ****kwargs** – Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns Return cumulative maximum of Series or DataFrame.

Return type *Series* or DataFrame

See also:

core.window.Expanding.max Similar functionality but ignores `NaN` values.

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.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [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. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
     A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

Notes

See pandas API documentation for pandas.DataFrame.cummax for more.

cummin(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

Parameters

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – The index or the name of the axis. 0 is equivalent to None or 'index'.
- **skipna** (*bool, default True*) – Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- ***args** – Additional keywords have no effect but might be accepted for compatibility with NumPy.
- ****kwargs** – Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns Return cumulative minimum of Series or DataFrame.

Return type *Series* or DataFrame

See also:

core.window.Expanding.min Similar functionality but ignores `NaN` values.

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.

DataFrame.cumprod Return cumulative product over DataFrame axis.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [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. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
     A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
     A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

Notes

See pandas API documentation for pandas.DataFrame.cummin for more.

cumprod(*axis=None*, *skipna=True*, **args*, ***kwargs*)
Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

Parameters

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – The index or the name of the axis. 0 is equivalent to None or 'index'.
- **skipna** (*bool, default True*) – Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- ***args** – Additional keywords have no effect but might be accepted for compatibility with NumPy.
- ****kwargs** – Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns Return cumulative product of Series or DataFrame.

Return type *Series* or DataFrame

See also:

core.window.Expanding.prod Similar functionality but ignores NaN values.

DataFrame.prod Return the product 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.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0     2.0
1     NaN
2    10.0
```

(continues on next page)

(continued from previous page)

```
3   -10.0
4    -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [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 product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
     A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
     A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

Notes

See pandas API documentation for pandas.DataFrame.cumprod for more.

cumsum(*axis=None*, *skipna=True*, **args*, ***kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

Parameters

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – The index or the name of the axis. 0 is equivalent to None or 'index'.
- **skipna** (*bool, default True*) – Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- ***args** – Additional keywords have no effect but might be accepted for compatibility with NumPy.
- ****kwargs** – Additional keywords have no effect but might be accepted for compatibility with NumPy.

Returns Return cumulative sum of Series or DataFrame.

Return type *Series* or DataFrame

See also:

core.window.Expanding.sum Similar functionality but ignores NaN values.

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.

Examples

Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
```

(continues on next page)

(continued from previous page)

```
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [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. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
     A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
     A    B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

Notes

See pandas API documentation for pandas.DataFrame.cumsum for more.

describe(*percentiles=None*, *include=None*, *exclude=None*, *datetime_is_numeric=False*)
Generate descriptive statistics.

Descriptive statistics include those 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-like of numbers, optional*) – The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.
- **include** (*'all', list-like of dtypes or None (default), optional*) – A white list of data types to include in the result. Ignored for `Series`. Here are the options:
  - 'all' : All columns of the input will be included in the output.
  - A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use `'category'`
  - None (default) : The result will include all numeric columns.
- **exclude** (*list-like of dtypes or None (default), optional,*) – A black list of data types to omit from the result. Ignored for `Series`. Here are the options:
  - A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use `'category'`
  - None (default) : The result will exclude nothing.
- **datetime_is_numeric** (*bool, default False*) – Whether to treat datetime dtypes as numeric. This affects statistics calculated for the column. For DataFrame input, this also controls whether datetime columns are included by default.

  New in version 1.1.0.

Returns Summary statistics of the Series or Dataframe provided.

Return type *Series* or DataFrame

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.

DataFrame.select_dtypes Subset of a DataFrame including/excluding columns based on their dtype.

Notes

See pandas API documentation for pandas.DataFrame.describe for more. For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, `50` and upper percentiles. By default the lower percentile is `25` and the upper percentile is `75`. The `50` percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The *include* and *exclude* parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

Examples

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...   np.datetime64("2000-01-01"),
...   np.datetime64("2010-01-01"),
...   np.datetime64("2010-01-01")
... ])
>>> s.describe(datetime_is_numeric=True)
count                      3
mean     2006-09-01 08:00:00
min      2000-01-01 00:00:00
25%      2004-12-31 12:00:00
50%      2010-01-01 00:00:00
75%      2010-01-01 00:00:00
max      2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
...                    'numeric': [1, 2, 3],
...                    'object': ['a', 'b', 'c']
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a `DataFrame` regardless of data type.

```
>>> df.describe(include='all')
       categorical  numeric object
count            3      3.0      3
unique           3      NaN      3
top              f      NaN      a
freq             1      NaN      1
mean           NaN      2.0    NaN
std            NaN      1.0    NaN
min            NaN      1.0    NaN
25%            NaN      1.5    NaN
50%            NaN      2.0    NaN
75%            NaN      2.5    NaN
max            NaN      3.0    NaN
```

Describing a column from a `DataFrame` by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
```

(continues on next page)

(continued from previous page)

```
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a `DataFrame` description.

```
>>> df.describe(include=[np.number])
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a `DataFrame` description.

```
>>> df.describe(include=[object])
       object
count       3
unique      3
top         a
freq        1
```

Including only categorical columns from a `DataFrame` description.

```
>>> df.describe(include=['category'])
       categorical
count            3
unique           3
top              d
freq             1
```

Excluding numeric columns from a `DataFrame` description.

```
>>> df.describe(exclude=[np.number])
       categorical object
count            3      3
unique           3      3
top              f      a
freq             1      1
```

Excluding object columns from a `DataFrame` description.

```
>>> df.describe(exclude=[object])
       categorical  numeric
count            3      3.0
```

(continues on next page)

(continued from previous page)

```
unique            3      NaN
top               f      NaN
freq              1      NaN
mean            NaN      2.0
std             NaN      1.0
min             NaN      1.0
25%             NaN      1.5
50%             NaN      2.0
75%             NaN      2.5
max             NaN      3.0
```

diff(*periods=1*, *axis=0*)

First discrete difference of element.

Calculates the difference of a Dataframe element compared with another element in the Dataframe (default is element in previous row).

Parameters

- **periods** (*int, default 1*) – Periods to shift for calculating difference, accepts negative values.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – Take difference over rows (0) or columns (1).

Returns First differences of the Series.

Return type Dataframe

See also:

Dataframe.pct_change Percent change over given number of periods.

Dataframe.shift Shift index by desired number of periods with an optional time freq.

Series.diff First discrete difference of object.

Notes

See pandas API documentation for pandas.DataFrame.diff for more. For boolean dtypes, this uses `operator.xor()` rather than `operator.sub()`. The result is calculated according to current dtype in Dataframe, however dtype of the result is always float64.

Examples

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b   c
0  1  1   1
1  2  1   4
2  3  2   9
```

(continues on next page)

(continued from previous page)

```
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(axis=1)
    a  b   c
0 NaN  0   0
1 NaN -1   3
2 NaN -1   7
3 NaN -1  13
4 NaN  0  20
5 NaN  2  28
```

Difference with 3rd previous row

```
>>> 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
```

Overflow in input dtype

```
>>> df = pd.DataFrame({'a': [1, 0]}, dtype=np.uint8)
>>> df.diff()
       a
0    NaN
1  255.0
```

div(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.truediv for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
```

(continues on next page)

(continued from previous page)

```
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

divide(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.truediv for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
```

(continues on next page)

(continued from previous page)

```
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

drop(*labels=None*, *axis=0*, *index=None*, *columns=None*, *level=None*, *inplace=False*, *errors='raise'*)

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level. See the *user guide <advanced.shown_levels>* for more information about the now unused levels.

Parameters

- **labels** (*single label or list-like*) – Index or column labels to drop.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – Whether to drop labels

from the index (0 or 'index') or columns (1 or 'columns').

- **index** (*single label or list-like*) – Alternative to specifying axis (`labels, axis=0` is equivalent to `index=labels`).
- **columns** (*single label or list-like*) – Alternative to specifying axis (`labels, axis=1` is equivalent to `columns=labels`).
- **level** (*int or level name, optional*) – For MultiIndex, level from which the labels will be removed.
- **inplace** (*bool, default False*) – If False, return a copy. Otherwise, do operation inplace and return None.
- **errors** (*{'ignore', 'raise'}, default 'raise'*) – If 'ignore', suppress error and only existing labels are dropped.

Returns DataFrame without the removed index or column labels or None if `inplace=True`.

Return type DataFrame or None

Raises **KeyError** – If any of the labels is not found in the selected axis.

See also:

DataFrame.loc Label-location based indexer for selection by label.

DataFrame.dropna Return DataFrame with labels on given axis omitted where (all or any) data are missing.

DataFrame.drop_duplicates Return DataFrame with duplicate rows removed, optionally only considering certain columns.

Series.drop Return Series with specified index labels removed.

Examples

```
>>> df = pd.DataFrame(np.arange(12).reshape(3, 4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                             [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3, 0.2]])
>>> df
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
                big
lama    speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
        weight  1.0
        length  0.3
```

```
>>> df.drop(index='length', level=1)
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
cow     speed   30.0    20.0
        weight  250.0   150.0
falcon  speed   320.0   250.0
        weight  1.0     0.8
```

Notes

See pandas API documentation for pandas.DataFrame.drop for more.

drop_duplicates(*keep='first'*, *inplace=False*, ***kwargs*)
Return DataFrame with duplicate rows removed.

Considering certain columns is optional. Indexes, including time indexes are ignored.

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** (*bool, default False*) – Whether to drop duplicates in place or to return a copy.
- **ignore_index** (*bool, default False*) – If True, the resulting axis will be labeled 0, 1, …, n - 1.

  New in version 1.0.0.

Returns DataFrame with duplicates removed or None if `inplace=True`.

Return type DataFrame or None

See also:

DataFrame.value_counts Count unique combinations of columns.

Examples

Consider dataset containing ramen rating.

```
>>> df = pd.DataFrame({
...     'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
...     'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
...     'rating': [4, 4, 3.5, 15, 5]
... })
>>> df
    brand style  rating
0  Yum Yum   cup     4.0
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

By default, it removes duplicate rows based on all columns.

```
>>> df.drop_duplicates()
    brand style  rating
0  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

To remove duplicates on specific column(s), use `subset`.

```
>>> df.drop_duplicates(subset=['brand'])
    brand style  rating
0  Yum Yum   cup     4.0
2  Indomie   cup     3.5
```

To remove duplicates and keep last occurrences, use `keep`.

```
>>> df.drop_duplicates(subset=['brand', 'style'], keep='last')
    brand style  rating
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
4  Indomie  pack     5.0
```

Notes

See pandas API documentation for pandas.DataFrame.drop_duplicates for more.

droplevel(*level*, *axis=0*)

Return Series/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*) – Axis along which the level(s) is removed:
  - 0 or 'index': remove level(s) in column.
  - 1 or 'columns': remove level(s) in row.

Returns Series/DataFrame with requested index / column level(s) removed.

Return type Series/DataFrame

Examples

```
>>> df = pd.DataFrame([
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]).set_index([0, 1]).rename_axis(['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
```

Notes

See pandas API documentation for pandas.DataFrame.droplevel for more.

dropna(*axis=0*, *how='any'*, *thresh=None*, *subset=None*, *inplace=False*)

Remove missing values.

See the User Guide for more on which values are considered missing, and how to work with missing data.

Parameters

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – Determine if rows or columns which contain missing values are removed.
  - 0, or 'index' : Drop rows which contain missing values.
  - 1, or 'columns' : Drop columns which contain missing value.

  Changed in version 1.0.0: Pass tuple or list to drop on multiple axes. Only a single axis is allowed.
- **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 with NA entries dropped from it or None if `inplace=True`.

Return type DataFrame or None

See also:

DataFrame.isna Indicate missing values.

DataFrame.notna Indicate existing (non-missing) values.

DataFrame.fillna Replace missing values.

Series.dropna Drop missing values.

Index.dropna Drop missing indices.

Examples

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

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        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

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        NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'toy'])
       name        toy       born
```

(continues on next page)

(continued from previous page)

```
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Keep the DataFrame with valid entries in the same variable.

```
>>> df.dropna(inplace=True)
>>> df
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Notes

See pandas API documentation for pandas.DataFrame.dropna for more.

eq(*other*, *axis='columns'*, *level=None*)

Get Equal to of dataframe and other, element-wise (binary operator *eq*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 'columns'*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns').
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns Result of the comparison.

Return type DataFrame of bool

See also:

DataFrame.eq Compare DataFrames for equality elementwise.

DataFrame.ne Compare DataFrames for inequality elementwise.

DataFrame.le Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

See pandas API documentation for pandas.DataFrame.eq for more. Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
```

(continues on next page)

(continued from previous page)

```
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

ewm(*com=None*, *span=None*, *halflife=None*, *alpha=None*, *min_periods=0*, *adjust=True*, *ignore_na=False*, *axis=0*, *times=None*)

Provide exponential weighted (EW) functions.

Available EW functions: `mean()`, `var()`, `std()`, `corr()`, `cov()`.

Exactly one parameter: `com`, `span`, `halflife`, or `alpha` must be provided.

Parameters

- **com** (*float, optional*) – Specify decay in terms of center of mass, $\alpha = 1/(1 + com)$, for $com \geq 0$.
- **span** (*float, optional*) – Specify decay in terms of span, $\alpha = 2/(span + 1)$, for $span \geq 1$.
- **halflife** (*float, str, timedelta, optional*) – Specify decay in terms of half-life, $\alpha = 1 - \exp(-\ln(2)/halflife)$, for $halflife > 0$.

  If `times` is specified, the time unit (str or timedelta) over which an observation decays to half its value. Only applicable to `mean()` and halflife value will not apply to the other functions.

  New in version 1.1.0.
- **alpha** (*float, optional*) – Specify smoothing factor α directly, $0 < \alpha \leq 1$.
- **min_periods** (*int, default 0*) – Minimum number of observations in window required to have a value (otherwise result is NA).
- **adjust** (*bool, default True*) – Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average).
  - When `adjust=True` (default), the EW function is calculated using weights $w_i = (1 - \alpha)^i$. For example, the EW moving average of the series $[x_0, x_1, ..., x_t]$ would be:

$$y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2 x_{t-2} + ... + (1 - \alpha)^t x_0}{1 + (1 - \alpha) + (1 - \alpha)^2 + ... + (1 - \alpha)^t}$$

  - When `adjust=False`, the exponentially weighted function is calculated recursively:

$$y_0 = x_0$$
$$y_t = (1 - \alpha)y_{t-1} + \alpha x_t,$$

- **ignore_na** (*bool, default False*) – Ignore missing values when calculating weights; specify `True` to reproduce pre-0.15.0 behavior.
  - When `ignore_na=False` (default), weights are based on absolute positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of [x_0, None, x_2] are $(1 - \alpha)^2$ and 1 if `adjust=True`, and $(1 - \alpha)^2$ and α if `adjust=False`.
  - When `ignore_na=True` (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x_0 and x_2 used in calculating the final weighted average of [x_0, None, x_2] are $1 - \alpha$ and 1 if `adjust=True`, and $1 - \alpha$ and α if `adjust=False`.
- **axis** (*{0, 1}, default 0*) – The axis to use. The value 0 identifies the rows, and 1 identifies the columns.

- **times** (*str, np.ndarray, Series, default None*) – New in version 1.1.0.

  Times corresponding to the observations. Must be monotonically increasing and `datetime64[ns]` dtype.

  If str, the name of the column in the DataFrame representing the times.

  If 1-D array like, a sequence with the same shape as the observations.

  Only applicable to `mean()`.

Returns A Window sub-classed for the particular operation.

Return type DataFrame

See also:

rolling Provides rolling window calculations.

expanding Provides expanding transformations.

Notes

See pandas API documentation for pandas.DataFrame.ewm for more.

More details can be found at: Exponentially weighted windows.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

Specifying `times` with a timedelta `halflife` when computing mean.

```
>>> times = ['2020-01-01', '2020-01-03', '2020-01-10', '2020-01-15', '2020-01-
↪17']
>>> df.ewm(halflife='4 days', times=pd.DatetimeIndex(times)).mean()
          B
0  0.000000
1  0.585786
2  1.523889
```

(continues on next page)

(continued from previous page)

```
3  1.523889
4  3.233686
```

expanding(*min_periods=1*, *center=None*, *axis=0*, *method='single'*)

Provide expanding transformations.

Parameters

- **min_periods** (*int, default 1*) – Minimum number of observations in window required to have a value (otherwise result is NA).
- **center** (*bool, default False*) – Set the labels at the center of the window.
- **axis** (*int or str, default 0*) –
- **method** (*str {'single', 'table'}, default 'single'*) – Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

  This argument is only implemented when specifying `engine='numba'` in the method call.

  New in version 1.3.0.

Returns

Return type a Window sub-classed for the particular operation

See also:

rolling Provides rolling window calculations.

ewm Provides exponential weighted functions.

Notes

See pandas API documentation for pandas.DataFrame.expanding for more. By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

Examples

```
>>> df = pd.DataFrame({"B": [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

explode(*column*, *ignore_index: modin.pandas.base.BasePandasDataset.bool = False*)

Transform each element of a list-like to a row, replicating index values.

New in version 0.25.0.

Parameters

- **column** (*IndexLabel*) – Column(s) to explode. For multiple columns, specify a non-empty list with each element be str or tuple, and all specified columns their list-like data on same row of the frame must have matching length.

  New in version 1.3.0: Multi-column explode

- **ignore_index** (*bool, default False*) – If True, the resulting index will be labeled 0, 1, …, n - 1.

  New in version 1.1.0.

Returns Exploded lists to rows of the subset columns; index will be duplicated for these rows.

Return type DataFrame

Raises **ValueError :** –

- If columns of the frame are not unique. * If specified columns to explode is empty list. * If specified columns to explode have not matching count of elements rowwise in the frame.

See also:

DataFrame.unstack Pivot a level of the (necessarily hierarchical) index labels.

DataFrame.melt Unpivot a DataFrame from wide format to long format.

Series.explode Explode a DataFrame from list-like columns to long format.

Notes

See pandas API documentation for pandas.DataFrame.explode for more. This routine will explode list-likes including lists, tuples, sets, Series, and np.ndarray. The result dtype of the subset rows will be object. Scalars will be returned unchanged, and empty list-likes will result in a np.nan for that row. In addition, the ordering of rows in the output will be non-deterministic when exploding sets.

Examples

```
>>> df = pd.DataFrame({'A': [[0, 1, 2], 'foo', [], [3, 4]],
...                    'B': 1,
...                    'C': [['a', 'b', 'c'], np.nan, [], ['d', 'e']]})
>>> df
           A  B          C
0  [0, 1, 2]  1  [a, b, c]
1        foo  1        NaN
2         []  1         []
3     [3, 4]  1     [d, e]
```

Single-column explode.

```
>>> df.explode('A')
     A  B          C
0    0  1  [a, b, c]
0    1  1  [a, b, c]
0    2  1  [a, b, c]
1  foo  1        NaN
2  NaN  1         []
3    3  1     [d, e]
3    4  1     [d, e]
```

Multi-column explode.

```
>>> df.explode(list('AC'))
     A  B    C
0    0  1    a
0    1  1    b
0    2  1    c
1  foo  1  NaN
2  NaN  1  NaN
3    3  1    d
3    4  1    e
```

ffill(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)

Synonym for `DataFrame.fillna()` with `method='ffill'`.

Returns Object with missing values filled or None if `inplace=True`.

Return type Series/DataFrame or None

Notes

See pandas API documentation for pandas.DataFrame.pad for more.

filter(*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset the dataframe rows or columns according to the specified index labels.

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** (*str*) – Keep labels from axis for which "like in label == True".
- **regex** (*str (regular expression)*) – Keep labels from axis for which re.search(regex, label) == True.
- **axis** (*{0 or 'index', 1 or 'columns', None}, default None*) – The axis to filter on, expressed either as an index (int) or axis name (str). By default this is the info axis, 'index' for Series, 'columns' for DataFrame.

Returns

Return type same type as input object

See also:

DataFrame.loc Access a group of rows and columns by label(s) or a boolean array.

Notes

See pandas API documentation for pandas.DataFrame.filter for more. 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 = pd.DataFrame(np.array(([1, 2, 3], [4, 5, 6])),
...                   index=['mouse', 'rabbit'],
...                   columns=['one', 'two', 'three'])
>>> df
        one  two  three
mouse     1    2      3
rabbit    4    5      6
```

```
>>> # 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
```

first(*offset*)

Select initial 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, DateOffset or dateutil.relativedelta*) – The offset length of the data that will be selected. For instance, '1M' will display all the rows having their index within the first month.

Returns A subset of the caller.

Return type *Series* or DataFrame

Raises **TypeError** – If the index is not a `DatetimeIndex`

See also:

last Select final periods of time series based on a date offset.

at_time Select values at a particular time of the day.

between_time Select values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
            A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.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 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

Notes

See pandas API documentation for pandas.DataFrame.first for more.

first_valid_index()

Return index for first non-NA value or None, if no NA value is found.

Returns **scalar**

Return type type of index

Notes

See pandas API documentation for pandas.DataFrame.first_valid_index for more. If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

property flags

Get the properties associated with this pandas object.

The available flags are

- `Flags.allows_duplicate_labels`

See also:

Flags Flags that apply to pandas objects.

DataFrame.attrs Global metadata applying to this dataset.

Notes

See pandas API documentation for pandas.DataFrame.flags for more. "Flags" differ from "metadata". Flags reflect properties of the pandas object (the Series or DataFrame). Metadata refer to properties of the dataset, and should be stored in `DataFrame.attrs`.

Examples

```
>>> df = pd.DataFrame({"A": [1, 2]})
>>> df.flags
<Flags(allows_duplicate_labels=True)>
```

Flags can be get or set using `.`

```
>>> df.flags.allows_duplicate_labels
True
>>> df.flags.allows_duplicate_labels = False
```

Or by slicing with a key

```
>>> df.flags["allows_duplicate_labels"]
False
>>> df.flags["allows_duplicate_labels"] = True
```

floordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rfloordiv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.floordiv for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
```

(continues on next page)

(continued from previous page)

```
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

ge(*other*, *axis='columns'*, *level=None*)

Get Greater than or equal to of dataframe and other, element-wise (binary operator *ge*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!=*, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 'columns'*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns').
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns Result of the comparison.

Return type DataFrame of bool

See also:

DataFrame.eq Compare DataFrames for equality elementwise.

DataFrame.ne Compare DataFrames for inequality elementwise.

DataFrame.le Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

See pandas API documentation for pandas.DataFrame.ge for more. Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
```

(continues on next page)

(continued from previous page)

```
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

get(*key*, *default=None*)
Get item from object for given key (ex: DataFrame column).

Returns default value if not found.

Parameters **key** (*object*) –

Returns **value**

Return type same type as items contained in object

Notes

See pandas API documentation for pandas.DataFrame.get for more.

gt(*other*, *axis='columns'*, *level=None*)
Get Greater than of dataframe and other, element-wise (binary operator *gt*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!=*, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 'columns'*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns').
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns Result of the comparison.

Return type DataFrame of bool

See also:

DataFrame.eq Compare DataFrames for equality elementwise.

DataFrame.ne Compare DataFrames for inequality elementwise.

DataFrame.le Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

See pandas API documentation for pandas.DataFrame.gt for more. Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

head(*n=5*)

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.

For negative values of *n*, this function returns all rows except the last *n* rows, equivalent to `df[:-n]`.

Parameters **n** (*int, default 5*) – Number of rows to select.

Returns The first *n* rows of the caller object.

Return type same type as caller

See also:

DataFrame.tail Returns the last *n* rows.

Examples

```
>>> df = pd.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
```

For negative values of *n*

```
>>> df.head(-3)
      animal
0  alligator
1        bee
2     falcon
3       lion
```

(continues on next page)

(continued from previous page)

```
4     monkey
5     parrot
```

Notes

See pandas API documentation for pandas.DataFrame.head for more.

property 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 **IndexError** – When integer position is out of bounds.

See also:

DataFrame.at Access a single value for a row/column label pair.

DataFrame.loc Access a group of rows and columns by label(s).

DataFrame.iloc Access a group of rows and columns by integer position(s).

Examples

```
>>> df = pd.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
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

Notes

See pandas API documentation for pandas.DataFrame.iat for more.

idxmax(*axis=0*, *skipna=True*)

Return index of first occurrence of maximum over requested axis.

NA/null values are excluded.

Parameters

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.
- **skipna** (*bool, default True*) – Exclude NA/null values. If an entire row/column is NA, the result will be NA.

Returns Indexes of maxima along the specified axis.

Return type *Series*

Raises **ValueError** –

- If the row/column is empty

See also:

Series.idxmax Return index of the maximum element.

Notes

See pandas API documentation for pandas.DataFrame.idxmax for more. This method is the DataFrame version of `ndarray.argmax`.

Examples

Consider a dataset containing food consumption in Argentina.

```
>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
...                    'co2_emissions': [37.2, 19.66, 1712]},
...                    index=['Pork', 'Wheat Products', 'Beef'])
```

```
>>> df
                consumption  co2_emissions
Pork                  10.51         37.20
Wheat Products       103.11         19.66
Beef                  55.48       1712.00
```

By default, it returns the index for the maximum value in each column.

```
>>> df.idxmax()
consumption     Wheat Products
co2_emissions             Beef
dtype: object
```

To return the index for the maximum value in each row, use `axis="columns"`.

```
>>> df.idxmax(axis="columns")
Pork              co2_emissions
Wheat Products     consumption
Beef              co2_emissions
dtype: object
```

idxmin(*axis=0*, *skipna=True*)

Return index of first occurrence of minimum over requested axis.

NA/null values are excluded.

Parameters

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.
- **skipna** (*bool, default True*) – Exclude NA/null values. If an entire row/column is NA, the result will be NA.

Returns Indexes of minima along the specified axis.

Return type *Series*

Raises **ValueError** –

- If the row/column is empty

See also:

Series.idxmin Return index of the minimum element.

Notes

See pandas API documentation for pandas.DataFrame.idxmin for more. This method is the DataFrame version of `ndarray.argmin`.

Examples

Consider a dataset containing food consumption in Argentina.

```
>>> df = pd.DataFrame({'consumption': [10.51, 103.11, 55.48],
...                    'co2_emissions': [37.2, 19.66, 1712]},
...                    index=['Pork', 'Wheat Products', 'Beef'])
```

```
>>> df
                consumption  co2_emissions
Pork                  10.51         37.20
Wheat Products       103.11         19.66
Beef                  55.48       1712.00
```

By default, it returns the index for the minimum value in each column.

```
>>> df.idxmin()
consumption                Pork
co2_emissions    Wheat Products
dtype: object
```

To return the index for the minimum value in each row, use `axis="columns"`.

```
>>> df.idxmin(axis="columns")
Pork                consumption
Wheat Products    co2_emissions
Beef                consumption
dtype: object
```

property 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 boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series or DataFrame) 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 more at Selection by Position.

See also:

DataFrame.iat Fast integer location scalar accessor.

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 = pd.DataFrame(mydict)
>>> df
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

Indexing just the rows

With a scalar integer.

```
>>> type(df.iloc[0])
<class 'pandas.core.series.Series'>
>>> df.iloc[0]
a    1
b    2
c    3
d    4
Name: 0, dtype: int64
```

With a list of integers.

```
>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
>>> type(df.iloc[[0]])
<class 'pandas.core.frame.DataFrame'>
```

```
>>> df.iloc[[0, 1]]
     a    b    c    d
0    1    2    3    4
1  100  200  300  400
```

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
```

With a boolean mask the same length as the index.

```
>>> df.iloc[[True, False, True]]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

With a callable, useful in method chains. The *x* passed to the `lambda` is the DataFrame being sliced. This selects the rows whose index label even.

```
>>> df.iloc[lambda x: x.index % 2 == 0]
      a     b     c     d
0     1     2     3     4
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[0, 1]
2
```

With lists of integers.

```
>>> df.iloc[[0, 2], [1, 3]]
      b     d
0     2     4
2  2000  4000
```

With *slice* objects.

```
>>> df.iloc[1:3, 0:3]
      a     b     c
1   100   200   300
2  1000  2000  3000
```

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
```

With a callable function that expects the Series or DataFrame.

```
>>> df.iloc[:, lambda df: [0, 2]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

Notes

See pandas API documentation for pandas.DataFrame.iloc for more.

property index

Get the index for this DataFrame.

Returns The union of all indexes across the partitions.

Return type pandas.Index

infer_objects()

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

Returns converted

Return type same type as input object

See also:

to_datetime Convert argument to datetime.

to_timedelta Convert argument to timedelta.

to_numeric Convert argument to numeric type.

convert_dtypes Convert argument to best possible dtype.

Examples

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

Notes

See pandas API documentation for pandas.DataFrame.infer_objects for more.

isin(*values*)

Whether each element in the DataFrame is contained in values.

Parameters **values** (*iterable, Series, DataFrame or dict*) – The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dict, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

Returns DataFrame of booleans showing whether each element in the DataFrame is contained in values.

Return type DataFrame

See also:

DataFrame.eq Equality test for DataFrame.

Series.isin Equivalent method on Series.

Series.str.contains Test if pattern or regex is contained within a string of a Series or Index.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4], 'num_wings': [2, 0]},
...                   index=['falcon', 'dog'])
>>> 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
```

When `values` is a Series or DataFrame the index and column must match. Note that 'falcon' does not match based on the number of legs in df2.

```
>>> other = pd.DataFrame({'num_legs': [8, 2], 'num_wings': [0, 2]},
...                      index=['spider', 'falcon'])
>>> df.isin(other)
        num_legs  num_wings
falcon      True       True
dog        False      False
```

Notes

See pandas API documentation for pandas.DataFrame.isin for more.

isna()

Detect 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 Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

Return type DataFrame

See also:

DataFrame.isnull Alias of isna.

DataFrame.notna Boolean inverse of isna.

DataFrame.dropna Omit axes labels with missing values.

isna Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame(dict(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.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

Notes

See pandas API documentation for pandas.DataFrame.isna for more.

isnull()

Detect 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 Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

Return type DataFrame

See also:

DataFrame.isnull Alias of isna.

DataFrame.notna Boolean inverse of isna.

DataFrame.dropna Omit axes labels with missing values.

isna Top-level isna.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame(dict(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.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

Notes

See pandas API documentation for pandas.DataFrame.isna for more.

kurt(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased kurtosis over requested axis.

Kurtosis obtained 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.
- **skipna** (*bool, default True*) – Exclude NA/null values when computing the result.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **numeric_only** (*bool, default None*) – Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- ****kwargs** – Additional keyword arguments to be passed to the function.

Returns

Return type *Series* or DataFrame (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.kurt for more.

kurtosis(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased kurtosis over requested axis.

Kurtosis obtained 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.
- **skipna** (*bool, default True*) – Exclude NA/null values when computing the result.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **numeric_only** (*bool, default None*) – Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- ****kwargs** – Additional keyword arguments to be passed to the function.

Returns

Return type *Series* or DataFrame (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.kurt for more.

last(*offset*)

Select final periods of time series data based on a date offset.

For a DataFrame with a sorted DatetimeIndex, this function selects the last few rows based on a date offset.

Parameters **offset** (*str, DateOffset, dateutil.relativedelta*) – 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 A subset of the caller.

Return type *Series* or DataFrame

Raises **TypeError** – If the index is not a `DatetimeIndex`

See also:

first Select initial periods of time series based on a date offset.

at_time Select values at a particular time of the day.

between_time Select values between particular times of the day.

Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1, 2, 3, 4]}, index=i)
>>> ts
            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:

```
>>> ts.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.

Notes

See pandas API documentation for pandas.DataFrame.last for more.

last_valid_index()
Return index for last non-NA value or None, if no NA value is found.

Returns **scalar**

Return type type of index

Notes

See pandas API documentation for pandas.DataFrame.last_valid_index for more. If all elements are non-NA/null, returns None. Also returns None for empty Series/DataFrame.

le(*other*, *axis='columns'*, *level=None*)
Get Less than or equal to of dataframe and other, element-wise (binary operator *le*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!=*, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 'columns'*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns').
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns Result of the comparison.

Return type DataFrame of bool

See also:

DataFrame.eq Compare DataFrames for equality elementwise.

DataFrame.ne Compare DataFrames for inequality elementwise.

DataFrame.le Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

See pandas API documentation for pandas.DataFrame.le for more. Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
```

(continues on next page)

(continued from previous page)

```
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

property loc

Access a group of rows and columns by label(s) or a boolean array.

`.loc[]` is primarily label based, but may also be used with a boolean array.

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).
- A list or array of labels, e.g. `['a', 'b', 'c']`.
- A slice object with labels, e.g. `'a':'f'`.

  Warning: Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. `[True, False, True]`.
- An alignable boolean Series. The index of the key will be aligned before masking.
- An alignable Index. The Index of the returned selection will be the input.
- A `callable` function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above)

See more at Selection by Label.

Raises

- **KeyError** – If any items are not found.
- **IndexingError** – If an indexed key is passed and its index is unalignable to the frame index.

See also:

DataFrame.at Access a single value for a row/column label pair.

DataFrame.iloc Access group of rows and columns by integer position(s).

DataFrame.xs Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

Series.loc Access group of values using labels.

Examples

Getting values

```
>>> df = pd.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.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
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
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Alignable boolean Series:

```
>>> df.loc[pd.Series([False, True, False],
...        index=['viper', 'sidewinder', 'cobra'])]
            max_speed  shield
sidewinder          7       8
```

Index (same behavior as `df.reindex`)

```
>>> df.loc[pd.Index(["cobra", "viper"], name="foo")]
       max_speed  shield
foo
cobra          1       2
viper          4       5
```

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
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set 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
```

Set 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 rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...       index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
```

(continues on next page)

(continued from previous page)

```
   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
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
```

(continues on next page)

(continued from previous page)

```
shield       2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

Notes

See pandas API documentation for pandas.DataFrame.loc for more.

lt(*other*, *axis='columns'*, *level=None*)

Get Less than of dataframe and other, element-wise (binary operator *lt*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!=*, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 'columns'*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns').

- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns Result of the comparison.

Return type DataFrame of bool

See also:

DataFrame.eq Compare DataFrames for equality elementwise.

DataFrame.ne Compare DataFrames for inequality elementwise.

DataFrame.le Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

See pandas API documentation for pandas.DataFrame.lt for more. Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
```

(continues on next page)

(continued from previous page)

```
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
```

(continues on next page)

(continued from previous page)

```
...                                       ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

mad(*axis=None*, *skipna=None*, *level=None*)

Return the mean absolute deviation of the values over the requested axis.

Parameters

- **axis** (*{index (0), columns (1)}*) – Axis for the function to be applied on.
- **skipna** (*bool, default None*) – Exclude NA/null values when computing the result.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

Returns

Return type *Series* or DataFrame (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.mad for more.

max(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the maximum of the values over the requested axis.

If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

Parameters

- **axis** (*{index (0), columns (1)}*) – Axis for the function to be applied on.
- **skipna** (*bool, default True*) – Exclude NA/null values when computing the result.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

- **numeric_only** (*bool, default None*) – Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- ****kwargs** – Additional keyword arguments to be passed to the function.

Returns

Return type *Series* or DataFrame (if level specified)

See also:

Series.sum Return the sum.

Series.min Return the minimum.

Series.max Return the maximum.

Series.idxmin Return the index of the minimum.

Series.idxmax Return the index of the maximum.

DataFrame.sum Return the sum over the requested axis.

DataFrame.min Return the minimum over the requested axis.

DataFrame.max Return the maximum over the requested axis.

DataFrame.idxmin Return the index of the minimum over the requested axis.

DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.max()
8
```

Notes

See pandas API documentation for pandas.DataFrame.max for more.

mean(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the mean of the values over the requested axis.

Parameters

- **axis** (*{index (0), columns (1)}*) – Axis for the function to be applied on.
- **skipna** (*bool, default True*) – Exclude NA/null values when computing the result.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **numeric_only** (*bool, default None*) – Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- ****kwargs** – Additional keyword arguments to be passed to the function.

Returns

Return type *Series* or DataFrame (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.mean for more.

median(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return the median of the values over the requested axis.

Parameters

- **axis** (*{index (0), columns (1)}*) – Axis for the function to be applied on.
- **skipna** (*bool, default True*) – Exclude NA/null values when computing the result.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **numeric_only** (*bool, default None*) – Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- ****kwargs** – Additional keyword arguments to be passed to the function.

Returns

Return type *Series* or DataFrame (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.median for more.

memory_usage(*index=True*, *deep=False*)
Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of *object* dtype.

This value is displayed in *DataFrame.info* by default. This can be suppressed by setting `pandas.options.display.memory_usage` to False.

Parameters

- **index** (*bool, default True*) – Specifies whether to include the memory usage of the DataFrame's index in returned Series. If `index=True`, the memory usage of the index is the first item in the output.
- **deep** (*bool, default False*) – If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned values.

Returns A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

Return type *Series*

See also:

numpy.ndarray.nbytes Total bytes consumed by the elements of an ndarray.

Series.memory_usage Bytes consumed by a Series.

Categorical Memory-efficient array for string values with many repeated values.

DataFrame.info Concise summary of a DataFrame.

Examples

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000, dtype=int).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64            complex128  object  bool
0      1      1.0              1.0+0.0j       1  True
1      1      1.0              1.0+0.0j       1  True
2      1      1.0              1.0+0.0j       1  True
3      1      1.0              1.0+0.0j       1  True
4      1      1.0              1.0+0.0j       1  True
```

```
>>> df.memory_usage()
Index           128
int64         40000
float64       40000
complex128    80000
object        40000
```

(continues on next page)

(continued from previous page)

```
bool           5000
dtype: int64
```

```
>>> df.memory_usage(index=False)
int64         40000
float64       40000
complex128    80000
object        40000
bool           5000
dtype: int64
```

The memory footprint of *object* dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index            128
int64          40000
float64        40000
complex128     80000
object        180000
bool            5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5244
```

Notes

See pandas API documentation for pandas.DataFrame.memory_usage for more.

min(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)

Return the minimum of the values over the requested axis.

If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

Parameters

- **axis** (*{index (0), columns (1)}*) – Axis for the function to be applied on.
- **skipna** (*bool, default True*) – Exclude NA/null values when computing the result.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **numeric_only** (*bool, default None*) – Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- ****kwargs** – Additional keyword arguments to be passed to the function.

Returns

Return type *Series* or DataFrame (if level specified)

See also:

Series.sum Return the sum.

Series.min Return the minimum.

Series.max Return the maximum.

Series.idxmin Return the index of the minimum.

Series.idxmax Return the index of the maximum.

DataFrame.sum Return the sum over the requested axis.

DataFrame.min Return the minimum over the requested axis.

DataFrame.max Return the maximum over the requested axis.

DataFrame.idxmin Return the index of the minimum over the requested axis.

DataFrame.idxmax Return the index of the maximum over the requested axis.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.min()
0
```

Notes

See pandas API documentation for pandas.DataFrame.min for more.

mod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmod*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.

- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.mod for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
```

(continues on next page)

(continued from previous page)

```
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
```

(continues on next page)

(continued from previous page)

```
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

mode(*axis=0*, *numeric_only=False*, *dropna=True*)

Get the mode(s) of each element along the selected axis.

The mode of a set of values is the value that appears most often. It can be multiple values.

Parameters

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – The axis to iterate over while searching for the mode:
  - 0 or 'index' : get mode of each column
  - 1 or 'columns' : get mode of each row.
- **numeric_only** (*bool, default False*) – If True, only apply to numeric columns.
- **dropna** (*bool, default True*) – Don't consider counts of NaN/NaT.

Returns The modes of each column or row.

Return type DataFrame

See also:

Series.mode Return the highest frequency value in a Series.

Series.value_counts Return the counts of values in a Series.

Examples

```
>>> df = pd.DataFrame([('bird', 2, 2),
...                    ('mammal', 4, np.nan),
...                    ('arthropod', 8, 0),
...                    ('bird', 2, np.nan)],
...                   index=('falcon', 'horse', 'spider', 'ostrich'),
...                   columns=('species', 'legs', 'wings'))
>>> df
           species  legs  wings
falcon        bird     2    2.0
horse       mammal     4    NaN
spider   arthropod     8    0.0
ostrich       bird     2    NaN
```

By default, missing values are not considered, and the mode of wings are both 0 and 2. Because the resulting DataFrame has two rows, the second row of `species` and `legs` contains `NaN`.

```
>>> df.mode()
  species  legs  wings
0    bird   2.0    0.0
1     NaN   NaN    2.0
```

Setting `dropna=False` `NaN` values are considered and they can be the mode (like for wings).

```
>>> df.mode(dropna=False)
  species  legs  wings
0    bird     2    NaN
```

Setting `numeric_only=True`, only the mode of numeric columns is computed, and columns of other types are ignored.

```
>>> df.mode(numeric_only=True)
   legs  wings
0   2.0    0.0
1   NaN    2.0
```

To compute the mode over columns and not rows, use the axis parameter:

```
>>> df.mode(axis='columns', numeric_only=True)
           0    1
falcon   2.0  NaN
horse    4.0  NaN
spider   0.0  8.0
ostrich  2.0  NaN
```

Notes

See pandas API documentation for pandas.DataFrame.mode for more.

mul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.mul for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

multiply(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.mul for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
```

(continues on next page)

(continued from previous page)

```
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

ne(*other*, *axis='columns'*, *level=None*)

Get Not equal to of dataframe and other, element-wise (binary operator *ne*).

Among flexible wrappers (*eq*, *ne*, *le*, *lt*, *ge*, *gt*) to comparison operators.

Equivalent to ==, *!*=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 'columns'*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns').
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.

Returns Result of the comparison.

Return type DataFrame of bool

See also:

DataFrame.eq Compare DataFrames for equality elementwise.

DataFrame.ne Compare DataFrames for inequality elementwise.

DataFrame.le Compare DataFrames for less than inequality or equality elementwise.

DataFrame.lt Compare DataFrames for strictly less than inequality elementwise.

DataFrame.ge Compare DataFrames for greater than inequality or equality elementwise.

DataFrame.gt Compare DataFrames for strictly greater than inequality elementwise.

Notes

See pandas API documentation for pandas.DataFrame.ne for more. Mismatched indices will be unioned together. *NaN* values are considered different (i.e. *NaN* != *NaN*).

Examples

```
>>> df = pd.DataFrame({'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
```

(continues on next page)

(continued from previous page)

```
B  False    False
C   True    False
```

When *other* is a `Series`, the columns of a DataFrame are aligned with the index of *other* and broadcast:

```
>>> df != pd.Series([100, 250], index=["cost", "revenue"])
    cost  revenue
A   True     True
B   True    False
C  False     True
```

Use the method to control the broadcast axis:

```
>>> df.ne(pd.Series([100, 300], index=["A", "D"]), axis='index')
   cost  revenue
A  True    False
B  True     True
C  True     True
D  True     True
```

When comparing to an arbitrary sequence, the number of columns must match the number elements in *other*:

```
>>> df == [250, 100]
    cost  revenue
A   True     True
B  False    False
C  False    False
```

Use the method to control the axis:

```
>>> df.eq([250, 250, 100], axis='index')
    cost  revenue
A   True    False
B  False     True
C   True    False
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({'revenue': [300, 250, 100, 150]},
...                      index=['A', 'B', 'C', 'D'])
>>> other
   revenue
A      300
B      250
C      100
D      150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
```

(continues on next page)

(continued from previous page)

```
C  False     True
D  False    False
```

Compare to a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'cost': [250, 150, 100, 150, 300, 220],
...                              'revenue': [100, 250, 300, 200, 175, 225]},
...                             index=[['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2'],
...                                    ['A', 'B', 'C', 'A', 'B', 'C']])
>>> df_multindex
      cost  revenue
Q1 A   250      100
   B   150      250
   C   100      300
Q2 A   150      200
   B   300      175
   C   220      225
```

```
>>> df.le(df_multindex, level=1)
       cost  revenue
Q1 A   True     True
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

notna()

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 Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

Return type DataFrame

See also:

DataFrame.notnull Alias of notna.

DataFrame.isna Boolean inverse of notna.

DataFrame.dropna Omit axes labels with missing values.

notna Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame(dict(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.notna()
     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 = pd.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
```

Notes

See pandas API documentation for pandas.DataFrame.notna for more.

notnull()

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 Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

Return type DataFrame

See also:

DataFrame.notnull Alias of notna.

DataFrame.isna Boolean inverse of notna.

DataFrame.dropna Omit axes labels with missing values.

notna Top-level notna.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame(dict(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.notna()
     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 = pd.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
```

Notes

See pandas API documentation for pandas.DataFrame.notna for more.

nunique(*axis=0*, *dropna=True*)

Count number of distinct elements in specified axis.

Return Series with number of distinct elements. Can ignore NaN values.

Parameters

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – The axis to use. 0 or 'index' for row-wise, 1 or 'columns' for column-wise.
- **dropna** (*bool, default True*) – Don't include NaN in the counts.

Returns

Return type *Series*

See also:

Series.nunique Method nunique for Series.

DataFrame.count Count non-NA cells for each column or row.

Examples

```
>>> df = pd.DataFrame({'A': [4, 5, 6], 'B': [4, 1, 1]})
>>> df.nunique()
A    3
B    2
dtype: int64
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
dtype: int64
```

Notes

See pandas API documentation for pandas.DataFrame.nunique for more.

pad(*axis=None*, *inplace=False*, *limit=None*, *downcast=None*)

Synonym for `DataFrame.fillna()` with `method='ffill'`.

Returns Object with missing values filled or None if `inplace=True`.

Return type Series/DataFrame or None

Notes

See pandas API documentation for pandas.DataFrame.pad for more.

pct_change(*periods=1*, *fill_method='pad'*, *limit=None*, *freq=None*, ***kwargs*)
Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

Parameters

- **periods** (*int, default 1*) – Periods to shift for forming percent change.
- **fill_method** (*str, default 'pad'*) – How to handle NAs before computing percent changes.
- **limit** (*int, default None*) – The number of consecutive NAs to fill before stopping.
- **freq** (*DateOffset, timedelta, or str, optional*) – Increment to use from time series API (e.g. 'M' or BDay()).
- ****kwargs** – Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

Returns **chg** – The same type as the calling object.

Return type *Series* or DataFrame

See also:

Series.diff Compute the difference of two elements in a Series.

DataFrame.diff Compute the difference of two elements in a DataFrame.

Series.shift Shift the index by some number of periods.

DataFrame.shift Shift the index by some number of periods.

Examples

Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0         NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0         NaN
1         NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0         NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.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
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
...     '2014': [1371819, 41403351]},
...     index=['GOOG', 'APPL'])
>>> df
```

(continues on next page)

(continued from previous page)

```
          2016      2015      2014
GOOG   1769950   1500923   1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns', periods=-1)
          2016      2015  2014
GOOG  0.179241  0.094112   NaN
APPL -0.252395 -0.011860   NaN
```

Notes

See pandas API documentation for pandas.DataFrame.pct_change for more.

pipe(*func*, **args*, ***kwargs*)

Apply func(self, *args, **kwargs).

Parameters

- **func** (*function*) – Function to apply to the Series/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 Series/DataFrame.
- **args** (*iterable, optional*) – Positional arguments passed into `func`.
- **kwargs** (*mapping, optional*) – A dictionary of keyword arguments passed into `func`.

Returns **object**

Return type the return type of `func`.

See also:

DataFrame.apply Apply a function along input axis of DataFrame.

DataFrame.applymap Apply a function elementwise on a whole DataFrame.

Series.map Apply a mapping correspondence on a `Series`.

Notes

See pandas API documentation for pandas.DataFrame.pipe for more. Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> func(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...     .pipe(g, arg1=a)
...     .pipe(func, arg2=b, arg3=c)
... )
```

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 `arg2`:

```
>>> (df.pipe(h)
...    .pipe(g, arg1=a)
...    .pipe((func, 'arg2'), arg1=a, arg3=c)
...  )
```

pop(*item*)

Return item and drop from frame. Raise KeyError if not found.

Parameters **item** (*label*) – Label of column to be popped.

Returns

Return type *Series*

Examples

```
>>> df = pd.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
```

Notes

See pandas API documentation for pandas.DataFrame.pop for more.

pow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rpow*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.pow for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
```

(continues on next page)

(continued from previous page)

```
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

quantile(*q=0.5*, *axis=0*, *numeric_only=True*, *interpolation='linear'*)

Return values at the given quantile over requested axis.

Parameters

- **q** (*float or array-like, default 0.5 (50% quantile)*) – Value between 0 <= q <= 1, the quantile(s) to compute.
- **axis** (*{0, 1, 'index', 'columns'}, default 0*) – Equals 0 or 'index' for row-wise, 1 or 'columns' for column-wise.
- **numeric_only** (*bool, default True*) – If False, the quantile of datetime and

timedelta data will be computed as well.

- **interpolation** (*{'linear', 'lower', 'higher', 'midpoint', 'nearest'}*) – This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:
  - linear: *i + (j - i) * fraction*, where *fraction* is the fractional part of the index surrounded by *i* and *j*.
  - lower: *i*.
  - higher: *j*.
  - nearest: *i* or *j* whichever is nearest.
  - midpoint: (*i* + *j*) / 2.

Returns

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.

Return type *Series* or DataFrame

See also:

core.window.Rolling.quantile Rolling quantile.

numpy.percentile Numpy function to compute the percentile.

Examples

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
...                   columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
Name: 0.1, dtype: float64
>>> df.quantile([.1, .5])
       a     b
0.1  1.3   3.7
0.5  2.5  55.0
```

Specifying *numeric_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
...                    'B': [pd.Timestamp('2010'),
...                          pd.Timestamp('2011')],
...                    'C': [pd.Timedelta('1 days'),
...                          pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A                    1.5
B    2010-07-02 12:00:00
C        1 days 12:00:00
Name: 0.5, dtype: object
```

Notes

See pandas API documentation for pandas.DataFrame.quantile for more.

radd(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Get Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *radd*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.add for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

rank(*axis=0*, *method='average'*, *numeric_only=None*, *na_option='keep'*, *ascending=True*, *pct=False*)
Compute numerical data ranks (1 through n) along axis.

By default, equal values are assigned a rank that is the average of the ranks of those values.

Parameters

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – Index to direct ranking.
- **method** (*{'average', 'min', 'max', 'first', 'dense'}, default 'average'*) – How to rank the group of records that have the same value (i.e. ties):
  - average: average rank of the group
  - min: lowest rank in the group
  - max: highest rank in the group
  - first: ranks assigned in order they appear in the array
  - dense: like 'min', but rank always increases by 1 between groups.
- **numeric_only** (*bool, optional*) – For DataFrame objects, rank only numeric columns if set to True.
- **na_option** (*{'keep', 'top', 'bottom'}, default 'keep'*) – How to rank NaN values:
  - keep: assign NaN rank to NaN values
  - top: assign lowest rank to NaN values
  - bottom: assign highest rank to NaN values
- **ascending** (*bool, default True*) – Whether or not the elements should be ranked in ascending order.
- **pct** (*bool, default False*) – Whether or not to display the returned rankings in percentile form.

Returns Return a Series or DataFrame with data ranks as values.

Return type same type as caller

See also:

core.groupby.GroupBy.rank Rank of values within each group.

Examples

```
>>> df = pd.DataFrame(data={'Animal': ['cat', 'penguin', 'dog',
...                                    'spider', 'snake'],
...                         'Number_legs': [4, 2, 4, 8, np.nan]})
>>> df
    Animal  Number_legs
0      cat          4.0
1  penguin          2.0
2      dog          4.0
3   spider          8.0
4    snake          NaN
```

The following example shows how the method behaves with the above parameters:

- default_rank: this is the default behaviour obtained without using any parameter.
- max_rank: setting `method = 'max'` the records that have the same values are ranked using the highest rank (e.g.: since 'cat' and 'dog' are both in the 2nd and 3rd position, rank 3 is assigned.)
- NA_bottom: choosing `na_option = 'bottom'`, if there are records with NaN values they are placed at the bottom of the ranking.
- pct_rank: when setting `pct = True`, the ranking is expressed as percentile rank.

```
>>> df['default_rank'] = df['Number_legs'].rank()
>>> df['max_rank'] = df['Number_legs'].rank(method='max')
>>> df['NA_bottom'] = df['Number_legs'].rank(na_option='bottom')
>>> df['pct_rank'] = df['Number_legs'].rank(pct=True)
>>> df
    Animal  Number_legs  default_rank  max_rank  NA_bottom  pct_rank
0      cat          4.0           2.5       3.0        2.5     0.625
1  penguin          2.0           1.0       1.0        1.0     0.250
2      dog          4.0           2.5       3.0        2.5     0.625
3   spider          8.0           4.0       4.0        4.0     1.000
4    snake          NaN           NaN       NaN        5.0       NaN
```

Notes

See pandas API documentation for pandas.DataFrame.rank for more.

rdiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *truediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.rtruediv for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
```

(continues on next page)

(continued from previous page)

```
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

reindex(*index=None*, *columns=None*, *copy=True*, ***kwargs*)

Conform Series/DataFrame to new index with optional filling logic.

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

- **axes** (*keywords for*) – New labels / index to conform to, should be specified using keywords. Preferably an Index object to avoid duplicating data.
- **method** (*{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}*) – Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.
  - None (default): don't fill gaps
  - pad / ffill: Propagate last valid observation forward to next valid.
  - backfill / bfill: Use next valid observation to fill gap.
  - nearest: Use nearest valid observations to fill gap.
- **copy** (*bool, default True*) – Return a new object, even if the passed indexes are the same.
- **level** (*int or name*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*scalar, default np.NaN*) – Value to use for missing values. Defaults to NaN, but can be any "compatible" value.
- **limit** (*int, default None*) – Maximum number of consecutive elements to forward or backward fill.
- **tolerance** (*optional*) – Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation `abs(index[indexer] - target) <= tolerance`.

  Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

Returns

Return type Series/DataFrame with changed index.

See also:

DataFrame.set_index Set row labels.

DataFrame.reset_index Remove row labels or move them to new columns.

DataFrame.reindex_like Change to same indices as other DataFrame.

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 = pd.DataFrame({'http_status': [200, 200, 404, 404, 301],
...                    'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...                    index=index)
>>> 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)
               http_status  response_time
Safari               404.0           0.07
Iceweasel              NaN            NaN
Comodo Dragon          NaN            NaN
IE10                 404.0           0.08
Chrome               200.0           0.02
```

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the `NaN` values.

```
>>> df.reindex(new_index, fill_value=0)
               http_status  response_time
Safari                 404           0.07
Iceweasel                0           0.00
Comodo Dragon            0           0.00
```

(continues on next page)

(continued from previous page)

```
IE10                  404           0.08
Chrome                200           0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
              http_status response_time
Safari                404          0.07
Iceweasel         missing       missing
Comodo Dragon     missing       missing
IE10                  404          0.08
Chrome                200          0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         NaN
```

Or we can use "axis-style" keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox            200         NaN
Chrome             200         NaN
Safari             404         NaN
IE10               404         NaN
Konqueror          301         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 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
            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)
            prices
2009-12-29     NaN
```

(continues on next page)

(continued from previous page)

```
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
```

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to back-propagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the `method` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
            prices
2009-12-29   100.0
2009-12-30   100.0
2009-12-31   100.0
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
```

Please note that the `NaN` value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the `NaN` values present in the original dataframe, use the `fillna()` method.

See the user guide for more.

Notes

See pandas API documentation for pandas.DataFrame.reindex for more.

reindex_like(*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)

Return an object with matching indices as other object.

Conform the object to the same index on all axes. Optional filling logic, placing 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** (*Object of the same data type*) – Its row and column indices are used to define the new indices of this object.
- **method** (*{None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}*) – Method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- None (default): don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap.

- **copy** (*bool, default True*) – Return a new object, even if the passed indexes are the same.
- **limit** (*int, default None*) – Maximum number of consecutive labels to fill for inexact matches.
- **tolerance** (*optional*) – Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation `abs(index[indexer] - target) <= tolerance`.

  Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

Returns Same type as caller, but with changed indices on each axis.

Return type *Series* or DataFrame

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

See pandas API documentation for pandas.DataFrame.reindex_like for more. Same as calling `.reindex(index=other.index, columns=other.columns,...)`.

Examples

```
>>> df1 = pd.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 = pd.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
2014-02-13          30.0       low
2014-02-15          35.1    medium
```

```
>>> df2.reindex_like(df1)
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          28.0              NaN       low
2014-02-13          30.0              NaN       low
2014-02-14           NaN              NaN       NaN
2014-02-15          35.1              NaN    medium
```

rename_axis(*mapper=None*, *index=None*, *columns=None*, *axis=None*, *copy=True*, *inplace=False*)

Set the name of the axis for the index or columns.

Parameters

- **mapper** (*scalar, list-like, optional*) – Value to set the axis name attribute.
- **index** (*scalar, list-like, dict-like or function, optional*) – A scalar, list-like, dict-like or functions transformations to apply to that axis' values. Note that the `columns` parameter is not allowed if the object is a Series. This parameter only apply for DataFrame type objects.

  Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and/or `columns`.
- **columns** (*scalar, list-like, dict-like or function, optional*) – A scalar, list-like, dict-like or functions transformations to apply to that axis' values. Note that the `columns` parameter is not allowed if the object is a Series. This parameter only apply for DataFrame type objects.

  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.
- **copy** (*bool, default True*) – Also copy underlying data.
- **inplace** (*bool, default False*) – Modifies the object directly, instead of creating a new Series or DataFrame.

Returns The same type as the caller or None if `inplace=True`.

Return type *Series*, DataFrame, or None

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

See pandas API documentation for pandas.DataFrame.rename_axis for more. `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. In this case, the parameter `copy` is ignored.

The second calling convention will modify the names of the corresponding index if mapper is a list or a scalar. However, if mapper is dict-like or a function, it will use the deprecated behavior of modifying the axis *labels*.

We *highly* recommend using keyword arguments to clarify your intent.

Examples

Series

```
>>> s = pd.Series(["dog", "cat", "monkey"])
>>> s
0       dog
1       cat
2    monkey
dtype: object
>>> s.rename_axis("animal")
animal
0    dog
1    cat
2    monkey
dtype: object
```

DataFrame

```
>>> df = pd.DataFrame({"num_legs": [4, 4, 2],
...                    "num_arms": [0, 0, 2]},
...                   ["dog", "cat", "monkey"])
>>> df
        num_legs  num_arms
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("animal")
>>> df
        num_legs  num_arms
animal
dog            4         0
cat            4         0
monkey         2         2
>>> df = df.rename_axis("limbs", axis="columns")
```

(continues on next page)

(continued from previous page)

```
>>> df
limbs   num_legs  num_arms
animal
dog            4         0
cat            4         0
monkey         2         2
```

MultiIndex

```
>>> df.index = pd.MultiIndex.from_product([['mammal'],
...                                        ['dog', 'cat', 'monkey']],
...                                       names=['type', 'name'])
>>> df
limbs          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(index={'type': 'class'})
limbs          num_legs  num_arms
class  name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

```
>>> df.rename_axis(columns=str.upper)
LIMBS          num_legs  num_arms
type   name
mammal dog            4         0
       cat            4         0
       monkey         2         2
```

reorder_levels(*order*, *axis=0*)

Rearrange index levels using input order. May not drop or duplicate levels.

Parameters

- **order** (*list of int or list of str*) – List representing new level order. Reference level by number (position) or by key (label).
- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – Where to reorder levels.

Returns

Return type DataFrame

Notes

See pandas API documentation for pandas.DataFrame.reorder_levels for more.

resample(*rule*, *axis=0*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *base: Optional[int] = None*, *on=None*, *level=None*, *origin: Union[str, Timestamp, datetime.datetime, numpy.datetime64, int, numpy.int64, float] = 'start_day'*, *offset: Optional[Union[Timedelta, datetime.timedelta, numpy.timedelta64, int, numpy.int64, float, str]] = None*)

Resample time-series data.

Convenience method for frequency conversion and resampling of time series. The object must have a datetime-like index (*DatetimeIndex*, *PeriodIndex*, or *TimedeltaIndex*), or the caller must pass the label of a datetime-like series/index to the `on`/`level` keyword parameter.

Parameters

- **rule** (*DateOffset, Timedelta or str*) – The offset string or object representing target conversion.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – Which axis to use for up- or down-sampling. For *Series* this will default to 0, i.e. along the rows. Must be *DatetimeIndex*, *TimedeltaIndex* or *PeriodIndex*.
- **closed** (*{'right', 'left'}, default None*) – Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.
- **label** (*{'right', 'left'}, default None*) – Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.
- **convention** (*{'start', 'end', 's', 'e'}, default 'start'*) – For *PeriodIndex* only, controls whether to use the start or end of *rule*.
- **kind** (*{'timestamp', 'period'}, optional, default None*) – Pass 'timestamp' to convert the resulting index to a *DateTimeIndex* or 'period' to convert it to a *PeriodIndex*. By default the input representation is retained.
- **loffset** (*timedelta, default None*) – Adjust the resampled time labels.

  Deprecated since version 1.1.0: You should add the loffset to the *df.index* after the resample. See below.
- **base** (*int, default 0*) – For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0.

  Deprecated since version 1.1.0: The new arguments that you should use are 'offset' or 'origin'.
- **on** (*str, optional*) – For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.
- **level** (*str or int, optional*) – For a MultiIndex, level (name or number) to use for resampling. *level* must be datetime-like.
- **origin** (*{'epoch', 'start', 'start_day', 'end', 'end_day'}, Timestamp*) – or str, default 'start_day' The timestamp on which to adjust the grouping. The timezone of origin must match the timezone of the index. If a timestamp is not used, these values are also supported:
  - 'epoch': *origin* is 1970-01-01

- 'start': *origin* is the first value of the timeseries
- 'start_day': *origin* is the first day at midnight of the timeseries

New in version 1.1.0.

- 'end': *origin* is the last value of the timeseries
- 'end_day': *origin* is the ceiling midnight of the last day

New in version 1.3.0.

- **offset** (*Timedelta or str, default is None*) – An offset timedelta added to the origin.

New in version 1.1.0.

Returns Resampler object.

Return type pandas.core.Resampler

See also:

Series.resample Resample a Series.

DataFrame.resample Resample a DataFrame.

groupby Group DataFrame by mapping, function, label, or list of labels.

asfreq Reindex a DataFrame with the given frequency without grouping.

Notes

See pandas API documentation for pandas.DataFrame.resample for more. See the user guide for more.

To learn more about the offset strings, please see this link.

Examples

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00     3
2000-01-01 00:03:00    12
2000-01-01 00:06:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket `2000-01-01 00:03:00` contains the value 3, but the summed value in the resampled bucket with the label `2000-01-01 00:03:00` does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00     3
2000-01-01 00:06:00    12
2000-01-01 00:09:00    21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00     0
2000-01-01 00:03:00     6
2000-01-01 00:06:00    15
2000-01-01 00:09:00    15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5]   # Select first 5 rows
2000-01-01 00:00:00   0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00   2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the `NaN` values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(arraylike):
...     return np.sum(arraylike) + 5
...
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

Resample a year by quarter using 'start' *convention*. Values are assigned to the first quarter of the period.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
...                                             freq='A',
...                                             periods=2))
>>> s
2012    1
2013    2
Freq: A-DEC, dtype: int64
>>> s.resample('Q', convention='start').asfreq()
2012Q1    1.0
2012Q2    NaN
2012Q3    NaN
2012Q4    NaN
2013Q1    2.0
2013Q2    NaN
2013Q3    NaN
2013Q4    NaN
Freq: Q-DEC, dtype: float64
```

Resample quarters by month using 'end' *convention*. Values are assigned to the last month of the period.

```
>>> q = pd.Series([1, 2, 3, 4], index=pd.period_range('2018-01-01',
...                                                   freq='Q',
...                                                   periods=4))
>>> q
2018Q1    1
2018Q2    2
2018Q3    3
2018Q4    4
Freq: Q-DEC, dtype: int64
>>> q.resample('M', convention='end').asfreq()
2018-03    1.0
2018-04    NaN
2018-05    NaN
```

(continues on next page)

(continued from previous page)

```
2018-06    2.0
2018-07    NaN
2018-08    NaN
2018-09    3.0
2018-10    NaN
2018-11    NaN
2018-12    4.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword *on* can be used to specify the column instead of the index for resampling.

```
>>> d = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
...      'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
>>> df = pd.DataFrame(d)
>>> df['week_starting'] = pd.date_range('01/01/2018',
...                                     periods=8,
...                                     freq='W')
>>> df
   price  volume week_starting
0     10      50    2018-01-07
1     11      60    2018-01-14
2      9      40    2018-01-21
3     13     100    2018-01-28
4     14      50    2018-02-04
5     18     100    2018-02-11
6     17      40    2018-02-18
7     19      50    2018-02-25
>>> df.resample('M', on='week_starting').mean()
               price  volume
week_starting
2018-01-31     10.75    62.5
2018-02-28     17.00    60.0
```

For a DataFrame with MultiIndex, the keyword *level* can be used to specify on which level the resampling needs to take place.

```
>>> days = pd.date_range('1/1/2000', periods=4, freq='D')
>>> d2 = {'price': [10, 11, 9, 13, 14, 18, 17, 19],
...       'volume': [50, 60, 40, 100, 50, 100, 40, 50]}
>>> df2 = pd.DataFrame(
...     d2,
...     index=pd.MultiIndex.from_product(
...         [days, ['morning', 'afternoon']]
...     )
... )
>>> df2
                      price  volume
2000-01-01 morning       10      50
           afternoon     11      60
2000-01-02 morning        9      40
           afternoon     13     100
```

(continues on next page)

(continued from previous page)

```
2000-01-03 morning       14      50
           afternoon     18     100
2000-01-04 morning       17      40
           afternoon     19      50
>>> df2.resample('D', level=0).sum()
            price  volume
2000-01-01     21     110
2000-01-02     22     140
2000-01-03     32     150
2000-01-04     36      90
```

If you want to adjust the start of the bins based on a fixed timestamp:

```
>>> start, end = '2000-10-01 23:30:00', '2000-10-02 00:30:00'
>>> rng = pd.date_range(start, end, freq='7min')
>>> ts = pd.Series(np.arange(len(rng)) * 3, index=rng)
>>> ts
2000-10-01 23:30:00     0
2000-10-01 23:37:00     3
2000-10-01 23:44:00     6
2000-10-01 23:51:00     9
2000-10-01 23:58:00    12
2000-10-02 00:05:00    15
2000-10-02 00:12:00    18
2000-10-02 00:19:00    21
2000-10-02 00:26:00    24
Freq: 7T, dtype: int64
```

```
>>> ts.resample('17min').sum()
2000-10-01 23:14:00     0
2000-10-01 23:31:00     9
2000-10-01 23:48:00    21
2000-10-02 00:05:00    54
2000-10-02 00:22:00    24
Freq: 17T, dtype: int64
```

```
>>> ts.resample('17min', origin='epoch').sum()
2000-10-01 23:18:00     0
2000-10-01 23:35:00    18
2000-10-01 23:52:00    27
2000-10-02 00:09:00    39
2000-10-02 00:26:00    24
Freq: 17T, dtype: int64
```

```
>>> ts.resample('17min', origin='2000-01-01').sum()
2000-10-01 23:24:00     3
2000-10-01 23:41:00    15
2000-10-01 23:58:00    45
2000-10-02 00:15:00    45
Freq: 17T, dtype: int64
```

If you want to adjust the start of the bins with an *offset* Timedelta, the two following lines are equivalent:

```
>>> ts.resample('17min', origin='start').sum()
2000-10-01 23:30:00     9
2000-10-01 23:47:00    21
2000-10-02 00:04:00    54
2000-10-02 00:21:00    24
Freq: 17T, dtype: int64
```

```
>>> ts.resample('17min', offset='23h30min').sum()
2000-10-01 23:30:00     9
2000-10-01 23:47:00    21
2000-10-02 00:04:00    54
2000-10-02 00:21:00    24
Freq: 17T, dtype: int64
```

If you want to take the largest Timestamp as the end of the bins:

```
>>> ts.resample('17min', origin='end').sum()
2000-10-01 23:35:00     0
2000-10-01 23:52:00    18
2000-10-02 00:09:00    27
2000-10-02 00:26:00    63
Freq: 17T, dtype: int64
```

In contrast with the *start_day*, you can use *end_day* to take the ceiling midnight of the largest Timestamp as the end of the bins and drop the bins not containing data:

```
>>> ts.resample('17min', origin='end_day').sum()
2000-10-01 23:38:00     3
2000-10-01 23:55:00    15
2000-10-02 00:12:00    45
2000-10-02 00:29:00    45
Freq: 17T, dtype: int64
```

To replace the use of the deprecated *base* argument, you can now use *offset*, in this example it is equivalent to have *base=2*:

```
>>> ts.resample('17min', offset='2min').sum()
2000-10-01 23:16:00     0
2000-10-01 23:33:00     9
2000-10-01 23:50:00    36
2000-10-02 00:07:00    39
2000-10-02 00:24:00    24
Freq: 17T, dtype: int64
```

To replace the use of the deprecated *loffset* argument:

```
>>> from pandas.tseries.frequencies import to_offset
>>> loffset = '19min'
>>> ts_out = ts.resample('17min').sum()
>>> ts_out.index = ts_out.index + to_offset(loffset)
>>> ts_out
2000-10-01 23:33:00     0
2000-10-01 23:50:00     9
```

(continues on next page)

(continued from previous page)

```
2000-10-02 00:07:00    21
2000-10-02 00:24:00    54
2000-10-02 00:41:00    24
Freq: 17T, dtype: int64
```

reset_index(*level=None*, *drop=False*, *inplace=False*, *col_level=0*, *col_fill=''*)

Reset the index, or a level of it.

Reset the index of the DataFrame, and use the default one instead. If the DataFrame has a MultiIndex, this method can remove one or more levels.

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 with the new index or None if `inplace=True`.

Return type DataFrame or None

See also:

DataFrame.set_index Opposite of reset_index.

DataFrame.reindex Change to new indices or expand indices.

DataFrame.reindex_like Change to same indices as other DataFrame.

Examples

```
>>> df = pd.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, and a new sequential index is used:

```
>>> 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 = pd.DataFrame([(389.0, 'fly'),
...                    ( 24.0, 'fly'),
...                    ( 80.5, 'run'),
...                    (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  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         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  389.0     fly
parrot           bird   24.0     fly
lion           mammal   80.5     run
monkey         mammal    NaN    jump
```

Notes

See pandas API documentation for pandas.DataFrame.reset_index for more.

rfloordiv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *floordiv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value

before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.rfloordiv for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
```

(continues on next page)

(continued from previous page)

```
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

rmod(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *mod*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.rmod for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

rmul(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rmul*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.mul for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
```

(continues on next page)

(continued from previous page)

```
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

rolling(*window*, *min_periods=None*, *center=False*, *win_type=None*, *on=None*, *axis=0*, *closed=None*, *method='single'*)

Provide rolling window calculations.

Parameters

- **window** (*int, offset, or BaseIndexer subclass*) – Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

  If its an offset then this will be the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes.

  If a BaseIndexer subclass is passed, calculates the window boundaries based on the defined `get_window_bounds` method. Additional rolling keyword arguments, namely *min_periods*, *center*, and *closed* will be passed to *get_window_bounds*.

- **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.

- **center** (*bool, default False*) – Set the labels at the center of the window.

- **win_type** (*str, default None*) – Provide a window type. If `None`, all points are evenly weighted. See the notes below for further information.

- **on** (*str, optional*) – For a DataFrame, a datetime-like column or Index level on which to calculate the rolling window, rather than the DataFrame's index. Provided integer column is ignored and excluded from result since an integer index is not used to calculate the rolling window.

- **axis** (*int or str, default 0*) –

- **closed** (*str, default None*) – Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. Defaults to 'right'.

  Changed in version 1.2.0: The closed parameter with fixed windows is now supported.

- **method** (*str {'single', 'table'}, default 'single'*) – Execute the rolling operation per single column or row (`'single'`) or over the entire object (`'table'`).

  This argument is only implemented when specifying `engine='numba'` in the method call.

New in version 1.3.0.

Returns

Return type a Window or Rolling sub-classed for the particular operation

See also:

expanding Provides expanding transformations.

ewm Provides exponential weighted functions.

Notes

See pandas API documentation for pandas.DataFrame.rolling for more. By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

To learn more about the offsets & frequency strings, please see this link.

If `win_type=None`, all points are evenly weighted; otherwise, `win_type` can accept a string of any scipy.signal window function.

Certain Scipy window types require additional parameters to be passed in the aggregation function. The additional parameters must match the keywords specified in the Scipy window type method signature. Please see the third example below on how to add the additional parameters.

Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
     B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
     B
0  NaN
1  0.5
2  1.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, using the 'gaussian' window type (note how we need to specify std).

```
>>> df.rolling(2, win_type='gaussian').sum(std=3)
          B
0       NaN
1  0.986207
2  2.958621
```

(continues on next page)

(continued from previous page)

```
3       NaN
4       NaN
```

Rolling sum with a window length of 2, min_periods defaults to the window length.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the min_periods

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

Same as above, but with forward-looking windows

```
>>> indexer = pd.api.indexers.FixedForwardWindowIndexer(window_size=2)
>>> df.rolling(window=indexer, min_periods=1).sum()
     B
0  1.0
1  3.0
2  2.0
3  4.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                   index = [pd.Timestamp('20130101 09:00:00'),
...                            pd.Timestamp('20130101 09:00:02'),
...                            pd.Timestamp('20130101 09:00:03'),
...                            pd.Timestamp('20130101 09:00:05'),
...                            pd.Timestamp('20130101 09:00:06')])
```

```
>>> df
                     B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for min_periods is 1.

```
>>> df.rolling('2s').sum()
                      B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

round(*decimals=0*, **args*, ***kwargs*)

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.
- ***args** – Additional keywords have no effect but might be accepted for compatibility with numpy.
- ****kwargs** – Additional keywords have no effect but might be accepted for compatibility with numpy.

Returns A DataFrame with the affected columns rounded to the specified number of decimal places.

Return type DataFrame

See also:

numpy.around Round a numpy array to the given number of decimals.

Series.round Round a Series to the given number of decimals.

Examples

```
>>> df = pd.DataFrame([(.21, .32), (.01, .67), (.66, .03), (.21, .18)],
...                   columns=['dogs', 'cats'])
>>> df
    dogs  cats
0  0.21  0.32
1  0.01  0.67
2  0.66  0.03
3  0.21  0.18
```

By providing an integer each column is rounded to the same number of decimal places

```
>>> df.round(1)
    dogs  cats
0   0.2   0.3
1   0.0   0.7
2   0.7   0.0
3   0.2   0.2
```

With a dict, the number of places for specific columns can be specified with the column names as key and the number of decimal places as value

```
>>> df.round({'dogs': 1, 'cats': 0})
    dogs  cats
0   0.2   0.0
1   0.0   1.0
2   0.7   0.0
3   0.2   0.0
```

Using a Series, the number of places for specific columns can be specified with the column names as index and the number of decimal places as value

```
>>> decimals = pd.Series([0, 1], index=['cats', 'dogs'])
>>> df.round(decimals)
    dogs  cats
0   0.2   0.0
1   0.0   1.0
2   0.7   0.0
3   0.2   0.0
```

Notes

See pandas API documentation for pandas.DataFrame.round for more.

rpow(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *pow*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.rpow for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
```

(continues on next page)

(continued from previous page)

```
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

rsub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *sub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.rsub for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

rtruediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *truediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.rtruediv for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

sample(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*, *axis=None*, *ignore_index=False*)

Return a random sample of items from an axis of object.

You can use *random_state* for reproducibility.

Parameters

- **n** (*int, optional*) – Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.
- **frac** (*float, optional*) – Fraction of axis items to return. Cannot be used with *n*.
- **replace** (*bool, default False*) – Allow or disallow sampling of the same row more than once.
- **weights** (*str or ndarray-like, optional*) – Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when axis = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. Infinite values not allowed.
- **random_state** (*int, array-like, BitGenerator, np.random.RandomState, optional*) – If int, array-like, or BitGenerator (NumPy>=1.17), seed for random number generator If np.random.RandomState, use as numpy RandomState object.

  Changed in version 1.1.0: array-like and BitGenerator (for NumPy>=1.17) object now passed to np.random.RandomState() as seed
- **axis** (*{0 or 'index', 1 or 'columns', None}, default None*) – Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames).
- **ignore_index** (*bool, default False*) – If True, the resulting index will be labeled 0, 1, …, n - 1.

  New in version 1.3.0.

Returns A new object of same type as caller containing *n* items randomly sampled from the caller object.

Return type *Series* or DataFrame

See also:

DataFrameGroupBy.sample Generates random samples from each group of a DataFrame object.

SeriesGroupBy.sample Generates random samples from each group of a Series object.

numpy.random.choice Generates a random sample from a given 1-D numpy array.

Notes

See pandas API documentation for pandas.DataFrame.sample for more. If *frac* > 1, *replacement* should be set to *True*.

Examples

```
>>> df = pd.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'])
>>> df
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
dog            4          0                  2
spider         8          0                  1
fish           0          0                  8
```

Extract 3 random elements from the `Series` `df['num_legs']`: Note that we use *random_state* to ensure the reproducibility of the examples.

```
>>> df['num_legs'].sample(n=3, random_state=1)
fish      0
spider    8
falcon    2
Name: num_legs, dtype: int64
```

A random 50% sample of the `DataFrame` with replacement:

```
>>> df.sample(frac=0.5, replace=True, random_state=1)
      num_legs  num_wings  num_specimen_seen
dog          4          0                  2
fish         0          0                  8
```

An upsample sample of the `DataFrame` with replacement: Note that *replace* parameter has to be *True* for *frac* parameter > 1.

```
>>> df.sample(frac=2, replace=True, random_state=1)
        num_legs  num_wings  num_specimen_seen
dog            4          0                  2
fish           0          0                  8
falcon         2          2                 10
falcon         2          2                 10
fish           0          0                  8
dog            4          0                  2
fish           0          0                  8
dog            4          0                  2
```

Using a DataFrame column as weights. Rows with larger value in the *num_specimen_seen* column are more likely to be sampled.

```
>>> df.sample(n=2, weights='num_specimen_seen', random_state=1)
        num_legs  num_wings  num_specimen_seen
falcon         2          2                 10
fish           0          0                  8
```

sem(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

- **axis** (*{index (0), columns (1)}*) –
- **skipna** (*bool, default True*) – Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **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. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

Return type *Series* or DataFrame (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.sem for more. To have the same behaviour as *numpy.std*, use *ddof=0* (instead of the default *ddof=1*)

set_axis(*labels*, *axis=0*, *inplace=False*)
Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Parameters

- **labels** (*list-like, Index*) – The values for the new index.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – The axis to update. The value 0 identifies the rows, and 1 identifies the columns.
- **inplace** (*bool, default False*) – Whether to return a new DataFrame instance.

Returns **renamed** – An object of type DataFrame or None if `inplace=True`.

Return type DataFrame or None

See also:

DataFrame.rename_axis Alter the name of the index or columns. Examples ——— >>> df = pd.DataFrame({“A”: [1, 2, 3], “B”: [4, 5, 6]}) Change the row labels. >>> df.set_axis([‘a’, ‘b’, ‘c’], axis=’index’) A B a 1 4 b 2 5 c 3 6 Change the column labels. >>> df.set_axis([‘I’, ‘II’], axis=’columns’) I II 0 1 4 1 2 5 2 3 6 Now, update the labels inplace. >>> df.set_axis([‘i’, ‘ii’], axis=’columns’, inplace=True) >>> df i ii 0 1 4 1 2 5 2 3 6

Notes

See pandas API documentation for pandas.DataFrame.set_axis for more.

set_flags(*, *copy: modin.pandas.base.BasePandasDataset.bool = False*, *allows_duplicate_labels: Optional[modin.pandas.base.BasePandasDataset.bool] = None*)

Return a new object with updated flags.

Parameters **allows_duplicate_labels** (*bool, optional*) – Whether the returned object allows duplicate labels.

Returns The same type as the caller.

Return type *Series* or DataFrame

See also:

DataFrame.attrs Global metadata applying to this dataset.

DataFrame.flags Global flags applying to this object.

Notes

See pandas API documentation for pandas.DataFrame.set_flags for more. This method returns a new object that's a view on the same data as the input. Mutating the input or the output values will be reflected in the other.

This method is intended to be used in method chains.

"Flags" differ from "metadata". Flags reflect properties of the pandas object (the Series or DataFrame). Metadata refer to properties of the dataset, and should be stored in `DataFrame.attrs`.

Examples

```
>>> df = pd.DataFrame({"A": [1, 2]})
>>> df.flags.allows_duplicate_labels
True
>>> df2 = df.set_flags(allows_duplicate_labels=False)
>>> df2.flags.allows_duplicate_labels
False
```

shift(*periods=1*, *freq=None*, *axis=0*, *fill_value=NoDefault.no_default*)

Shift index by desired number of periods with an optional time *freq*.

When *freq* is not passed, shift the index without realigning the data. If *freq* is passed (in this case, the index must be date or datetime, or it will raise a *NotImplementedError*), the index will be increased using the periods and the *freq*. *freq* can be inferred when specified as "infer" as long as either freq or inferred_freq attribute is set in the index.

Parameters

- **periods** (*int*) – Number of periods to shift. Can be positive or negative.
- **freq** (*DateOffset, tseries.offsets, timedelta, or str, optional*) – Offset to use from the tseries module or time rule (e.g. 'EOM'). If *freq* is specified then the index values are shifted but the data is not realigned. That is, use *freq* if you would like to extend the index when shifting and preserve the original data. If *freq* is

specified as "infer" then it will be inferred from the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown.

- **axis** (*{0 or 'index', 1 or 'columns', None}, default None*) – Shift direction.
- **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. For datetime, timedelta, or period data, etc. `NaT` is used. For extension dtypes, `self.dtype.na_value` is used.

  Changed in version 1.1.0.

Returns Copy of input object, shifted.

Return type DataFrame

See also:

Index.shift Shift values of Index.

DatetimeIndex.shift Shift values of DatetimeIndex.

PeriodIndex.shift Shift values of PeriodIndex.

tshift Shift the time index, using the index's frequency if available.

Examples

```
>>> df = pd.DataFrame({"Col1": [10, 20, 15, 30, 45],
...                    "Col2": [13, 23, 18, 33, 48],
...                    "Col3": [17, 27, 22, 37, 52]},
...                   index=pd.date_range("2020-01-01", "2020-01-05"))
>>> df
            Col1  Col2  Col3
2020-01-01    10    13    17
2020-01-02    20    23    27
2020-01-03    15    18    22
2020-01-04    30    33    37
2020-01-05    45    48    52
```

```
>>> df.shift(periods=3)
            Col1  Col2  Col3
2020-01-01   NaN   NaN   NaN
2020-01-02   NaN   NaN   NaN
2020-01-03   NaN   NaN   NaN
2020-01-04  10.0  13.0  17.0
2020-01-05  20.0  23.0  27.0
```

```
>>> df.shift(periods=1, axis="columns")
            Col1  Col2  Col3
2020-01-01   NaN    10    13
2020-01-02   NaN    20    23
2020-01-03   NaN    15    18
2020-01-04   NaN    30    33
2020-01-05   NaN    45    48
```

```
>>> df.shift(periods=3, fill_value=0)
            Col1  Col2  Col3
2020-01-01     0     0     0
2020-01-02     0     0     0
2020-01-03     0     0     0
2020-01-04    10    13    17
2020-01-05    20    23    27
```

```
>>> df.shift(periods=3, freq="D")
            Col1  Col2  Col3
2020-01-04    10    13    17
2020-01-05    20    23    27
2020-01-06    15    18    22
2020-01-07    30    33    37
2020-01-08    45    48    52
```

```
>>> df.shift(periods=3, freq="infer")
            Col1  Col2  Col3
2020-01-04    10    13    17
2020-01-05    20    23    27
2020-01-06    15    18    22
2020-01-07    30    33    37
2020-01-08    45    48    52
```

Notes

See pandas API documentation for pandas.DataFrame.shift for more.

property 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.

See also:

ndarray.size Number of elements in the array.

Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

Notes

See pandas API documentation for pandas.DataFrame.size for more.

skew(*axis=None*, *skipna=None*, *level=None*, *numeric_only=None*, ***kwargs*)
Return unbiased skew over requested axis.

Normalized by N-1.

Parameters

- **axis** (*{index (0), columns (1)}*) – Axis for the function to be applied on.
- **skipna** (*bool, default True*) – Exclude NA/null values when computing the result.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **numeric_only** (*bool, default None*) – Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.
- ****kwargs** – Additional keyword arguments to be passed to the function.

Returns

Return type *Series* or DataFrame (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.skew for more.

sort_index(*axis=0*, *level=None*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na_position='last'*, *sort_remaining=True*, *ignore_index: modin.pandas.base.BasePandasDataset.bool = False*, *key: Optional[Callable[[Index], Union[Index, ExtensionArray, numpy.ndarray, Series]]] = None*)
Sort object by labels (along an axis).

Returns a new DataFrame sorted by label if *inplace* argument is `False`, otherwise updates the original DataFrame and returns None.

Parameters

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – The axis along which to sort. The value 0 identifies the rows, and 1 identifies the columns.
- **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** (*bool or list-like of bools, default True*) – Sort ascending vs. descending. When the index is a MultiIndex the sort direction can be controlled for each level individually.
- **inplace** (*bool, default False*) – If True, perform operation in-place.
- **kind** (*{'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'*) – Choice of sorting algorithm. See also `numpy.sort()` for more information. *mergesort* and *stable* are the only stable algorithms. For DataFrames, this option is only applied when sorting on a single column or label.
- **na_position** (*{'first', 'last'}, default 'last'*) – Puts NaNs at the beginning if *first*; *last* puts NaNs at the end. Not implemented for MultiIndex.

- **sort_remaining** (*bool, default True*) – If True and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level.
- **ignore_index** (*bool, default False*) – If True, the resulting axis will be labeled 0, 1, …, n - 1.

  New in version 1.0.0.
- **key** (*callable, optional*) – If not None, apply the key function to the index values before sorting. This is similar to the *key* argument in the builtin `sorted()` function, with the notable difference that this *key* function should be *vectorized*. It should expect an `Index` and return an `Index` of the same shape. For MultiIndex inputs, the key is applied *per level*.

  New in version 1.1.0.

Returns The original DataFrame sorted by the labels or None if `inplace=True`.

Return type DataFrame or None

See also:

Series.sort_index Sort Series by the index.

DataFrame.sort_values Sort DataFrame by the value.

Series.sort_values Sort Series by the value.

Examples

```
>>> df = pd.DataFrame([1, 2, 3, 4, 5], index=[100, 29, 234, 1, 150],
...                   columns=['A'])
>>> df.sort_index()
     A
1    4
29   2
100  1
150  5
234  3
```

By default, it sorts in ascending order, to sort in descending order, use `ascending=False`

```
>>> df.sort_index(ascending=False)
     A
234  3
150  5
100  1
29   2
1    4
```

A key function can be specified which is applied to the index before sorting. For a `MultiIndex` this is applied to each level separately.

```
>>> df = pd.DataFrame({"a": [1, 2, 3, 4]}, index=['A', 'b', 'C', 'd'])
>>> df.sort_index(key=lambda x: x.str.lower())
   a
A  1
```

(continues on next page)

(continued from previous page)

```
b  2
C  3
d  4
```

Notes

See pandas API documentation for pandas.DataFrame.sort_index for more.

sort_values(*by*, *axis=0*, *ascending=True*, *inplace: modin.pandas.base.BasePandasDataset.bool = False*, *kind='quicksort'*, *na_position='last'*, *ignore_index: modin.pandas.base.BasePandasDataset.bool = False*, *key: Optional[Callable[[Index], Union[Index, ExtensionArray, numpy.ndarray, Series]]] = None*)

Sort by the values along either axis.

Parameters **by** (*str or list of str*) – Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels.
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels.

axis [{0 or 'index', 1 or 'columns'}, default 0] Axis to be sorted.

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.

kind [{'quicksort', 'mergesort', 'heapsort', 'stable'}, default 'quicksort'] Choice of sorting algorithm. See also `numpy.sort()` for more information. *mergesort* and *stable* are the only stable algorithms. For DataFrames, this option is only applied when sorting on a single column or label.

na_position [{'first', 'last'}, default 'last'] Puts NaNs at the beginning if *first*; *last* puts NaNs at the end.

ignore_index [bool, default False] If True, the resulting axis will be labeled 0, 1, …, n - 1.

New in version 1.0.0.

key [callable, optional] Apply the key function to the values before sorting. This is similar to the *key* argument in the builtin `sorted()` function, with the notable difference that this *key* function should be *vectorized*. It should expect a `Series` and return a Series with the same shape as the input. It will be applied to each column in *by* independently.

New in version 1.1.0.

Returns DataFrame with sorted values or None if `inplace=True`.

Return type DataFrame or None

See also:

`DataFrame.sort_index` Sort a DataFrame by the index.

`Series.sort_values` Similar method for a Series.

Examples

```
>>> df = pd.DataFrame({
...     'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2': [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
...     'col4': ['a', 'B', 'c', 'D', 'e', 'F']
... })
>>> df
  col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
  col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
5    C     4     3    F
4    D     7     2    e
3  NaN     8     4    D
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
  col1  col2  col3 col4
1    A     1     1    B
0    A     2     0    a
2    B     9     9    c
5    C     4     3    F
4    D     7     2    e
3  NaN     8     4    D
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
  col1  col2  col3 col4
4    D     7     2    e
5    C     4     3    F
2    B     9     9    c
0    A     2     0    a
1    A     1     1    B
3  NaN     8     4    D
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
  col1  col2  col3 col4
```

(continues on next page)

(continued from previous page)

```
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
2    B     9     9    c
0    A     2     0    a
1    A     1     1    B
```

Sorting with a key function

```
>>> df.sort_values(by='col4', key=lambda col: col.str.lower())
   col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
3  NaN     8     4    D
4    D     7     2    e
5    C     4     3    F
```

Natural sort with the key argument, using the *natsort <https://github.com/SethMMorton/natsort>* package.

```
>>> df = pd.DataFrame({
...    "time": ['0hr', '128hr', '72hr', '48hr', '96hr'],
...    "value": [10, 20, 30, 40, 50]
... })
>>> df
    time  value
0    0hr     10
1  128hr     20
2   72hr     30
3   48hr     40
4   96hr     50
>>> from natsort import index_natsorted
>>> df.sort_values(
...    by="time",
...    key=lambda x: np.argsort(index_natsorted(df["time"]))
... )
    time  value
0    0hr     10
3   48hr     40
2   72hr     30
4   96hr     50
1  128hr     20
```

Notes

See pandas API documentation for pandas.DataFrame.sort_values for more.

std(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)
Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

- **axis** (*{index (0), columns (1)}*) –
- **skipna** (*bool, default True*) – Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **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. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

Return type *Series* or DataFrame (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.std for more. To have the same behaviour as *numpy.std*, use *ddof=0* (instead of the default *ddof=1*)

sub(*other*, *axis='columns'*, *level=None*, *fill_value=None*)
Get Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rsub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or ‘index’) or columns (1 or ‘columns’). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.sub for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

subtract(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rsub*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.sub for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...        axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
```

(continues on next page)

(continued from previous page)

```
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

swapaxes(*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately.

Returns **y**

Return type same as input

Notes

See pandas API documentation for pandas.DataFrame.swapaxes for more.

swaplevel(*i=- 2*, *j=- 1*, *axis=0*)

Swap levels i and j in a `MultiIndex`.

Default is to swap the two innermost levels of the index.

Parameters

- **i** (*int or str*) – Levels of the indices to be swapped. Can pass level name as string.
- **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* – ——— >>> df = pd.DataFrame(… {"Grade": ["A", "B", "A", "C"]}, … index=[… ["Final exam", "Final exam", "Coursework", "Coursework"], … ["History", "Geography", "History", "Geography"], … ["January", "February", "March", "April"], …], …) >>> df

  Grade

  Final exam History January A Geography February B

  Coursework History March A Geography April C

  In the following example, we will swap the levels of the indices. Here, we will swap the levels column-wise, but levels can be swapped row-wise in a similar manner. Note that column-wise is the default behaviour. By not supplying any arguments for i and j, we swap the last and second to last indices.

```
>>> df.swaplevel()
                                Grade
Final exam  January     History     A
            February    Geography   B
Coursework  March       History     A
            April       Geography   C
```

By supplying one argument, we can choose which index to swap the last index with. We can for example swap the first index with the last one as follows.

```
>>> df.swaplevel(0)
                                Grade
January     History     Final exam  A
February    Geography   Final exam  B
March       History     Coursework  A
April       Geography   Coursework  C
```

We can also define explicitly which indices we want to swap by supplying values for both i and j. Here, we for example swap the first and second indices.

```
>>> df.swaplevel(0, 1)
                                Grade
History     Final exam  January     A
Geography   Final exam  February    B
History     Coursework  March       A
Geography   Coursework  April       C
```

Notes

See pandas API documentation for pandas.DataFrame.swaplevel for more.

`tail`(*n=5*)

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 The last *n* rows of the caller object.

Return type type of caller

See also:

DataFrame.head The first *n* rows of the caller object.

Examples

```
>>> df = pd.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
```

Notes

See pandas API documentation for pandas.DataFrame.tail for more.

take(*indices*, *axis=0*, *is_copy=None*, ***kwargs*)
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.
- **is_copy** (*bool*) – Before pandas 1.0, `is_copy=False` can be specified to ensure that the return value is an actual copy. Starting with pandas 1.0, `take` always returns a copy, and the keyword is therefore deprecated.

  Deprecated since version 1.0.0.
- ****kwargs** – For compatibility with `numpy.take()`. Has no effect on the output.

Returns **taken** – An array-like containing the elements taken from the object.

Return type same type as caller

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 = pd.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])
     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
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])
     name   class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

Notes

See pandas API documentation for pandas.DataFrame.take for more.

to_clipboard(*excel=True*, *sep=None*, ***kwargs*)

Copy object to the system clipboard.

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

Parameters

- **excel** (*bool, default True*) – Produce output in a csv format for easy pasting into excel.
  - True, use the provided separator for csv pasting.
  - 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:

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

read_clipboard Read text from clipboard and pass to read_table.

Notes

See pandas API documentation for pandas.DataFrame.to_clipboard for more. Requirements for your platform.

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

Examples

Copy the contents of a DataFrame to the clipboard.

```
>>> df = pd.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
```

to_csv(*path_or_buf=None*, *sep=','*, *na_rep=''*, *float_format=None*, *columns=None*, *header=True*, *index=True*, *index_label=None*, *mode='w'*, *encoding=None*, *compression='infer'*, *quoting=None*, *quotechar='"'*, *line_terminator=None*, *chunksize=None*, *date_format=None*, *doublequote=True*, *escapechar=None*, *decimal='.'*, *errors: str = 'strict'*, *storage_options: Optional[Dict[str, Any]] = None*)

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

Parameters

- **path_or_buf** (*str or file handle, default None*) – File path or object, if None is provided the result is returned as a string. If a non-binary file object is passed, it should be opened with *newline=''*, disabling universal newlines. If a binary file object is passed, *mode* might need to contain a *'b'*.

  Changed in version 1.2.0: Support for binary file objects was introduced.

- **sep** (*str, default ','*) – String of length 1. Field delimiter for the output file.
- **na_rep** (*str, default ''*) – Missing data representation.
- **float_format** (*str, default None*) – Format string for floating point numbers.
- **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.
- **index** (*bool, default True*) – Write row names (index).

- **index_label** (*str or sequence, or False, default None*) – Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the object uses MultiIndex. If False do not print fields for index names. Use index_label=False for easier importing in R.
- **mode** (*str*) – Python write mode, default 'w'.
- **encoding** (*str, optional*) – A string representing the encoding to use in the output file, defaults to 'utf-8'. *encoding* is not supported if *path_or_buf* is a non-binary file object.
- **compression** (*str or dict, default 'infer'*) – If str, represents compression mode. If dict, value at 'method' is the compression mode. Compression mode may be any of the following possible values: {'infer', 'gzip', 'bz2', 'zip', 'xz', None}. If compression mode is 'infer' and *path_or_buf* is path-like, then detect compression mode from the following extensions: '.gz', '.bz2', '.zip' or '.xz'. (otherwise no compression). If dict given and mode is one of {'zip', 'gzip', 'bz2'}, or inferred as one of the above, other entries passed as additional compression options.

  Changed in version 1.0.0: May now be a dict with key 'method' as compression mode and other entries as additional compression options if compression mode is 'zip'.

  Changed in version 1.1.0: Passing compression options as keys in dict is supported for compression modes 'gzip' and 'bz2' as well as 'zip'.

  Changed in version 1.2.0: Compression is supported for binary file objects.

  Changed in version 1.2.0: Previous versions forwarded dict entries for 'gzip' to *gzip.open* instead of *gzip.GzipFile* which prevented setting *mtime*.
- **quoting** (*optional constant from csv module*) – Defaults to csv.QUOTE_MINIMAL. If you have set a *float_format* then floats are converted to strings and thus csv.QUOTE_NONNUMERIC will treat them as non-numeric.
- **quotechar** (*str, default '"'*) – String of length 1. Character used to quote fields.
- **line_terminator** (*str, optional*) – The newline character or character sequence to use in the output file. Defaults to *os.linesep*, which depends on the OS in which this method is called ('\n' for linux, '\r\n' for Windows, i.e.).
- **chunksize** (*int or None*) – Rows to write at a time.
- **date_format** (*str, default None*) – Format string for datetime objects.
- **doublequote** (*bool, default True*) – Control quoting of *quotechar* inside a field.
- **escapechar** (*str, default None*) – String of length 1. Character used to escape *sep* and *quotechar* when appropriate.
- **decimal** (*str, default '.'*) – Character recognized as decimal separator. E.g. use ',' for European data.
- **errors** (*str, default 'strict'*) – Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

  New in version 1.1.0.
- **storage_options** (*dict, optional*) – Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec`. Please see `fsspec` and `urllib` for more details.

New in version 1.2.0.

Returns If path_or_buf is None, returns the resulting csv format as a string. Otherwise returns None.

Return type None or str

See also:

read_csv Load a CSV file into a DataFrame.

to_excel Write DataFrame to an Excel file.

Examples

```
>>> df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
...                    'mask': ['red', 'purple'],
...                    'weapon': ['sai', 'bo staff']})
>>> df.to_csv(index=False)
'name,mask,weapon\nRaphael,red,sai\nDonatello,purple,bo staff\n'
```

Create 'out.zip' containing 'out.csv'

```
>>> compression_opts = dict(method='zip',
...                         archive_name='out.csv')
>>> df.to_csv('out.zip', index=False,
...           compression=compression_opts)
```

Notes

See pandas API documentation for pandas.DataFrame.to_csv for more.

to_dict(*orient='dict'*, *into=<class 'dict'>*)

Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

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 Return a collections.abc.Mapping object representing the DataFrame. The resulting transformation depends on the *orient* parameter.

Return type dict, list or collections.abc.Mapping

See also:

DataFrame.from_dict Create a DataFrame from a dictionary.

DataFrame.to_json Convert a DataFrame to JSON format.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                    'col2': [0.5, 0.75]},
...                   index=['row1', 'row2'])
>>> df
      col1  col2
row1     1  0.50
row2     2  0.75
>>> df.to_dict()
{'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 0.5, 'row2': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': row1    1
         row2    2
Name: col1, dtype: int64,
'col2': row1    0.50
        row2    0.75
Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1, 'col2': 0.5}, {'col1': 2, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'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'>, {'col1': 1, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2, 'col2': 0.75})]
```

Notes

See pandas API documentation for pandas.DataFrame.to_dict for more.

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*, *storage_options: Optional[Dict[str, Any]] = None*)

Write object to an Excel sheet.

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** (*path-like, file-like, 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`.

  Deprecated since version 1.2.0: As the xlwt package is no longer maintained, the `xlwt` engine will be removed in a future version of pandas.
- **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.
- **storage_options** (*dict, optional*) – Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec`. Please see `fsspec` and `urllib` for more details.

  New in version 1.2.0.

See also:

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

ExcelWriter Class for writing DataFrame objects into excel sheets.

read_excel Read an Excel file into a pandas DataFrame.

read_csv Read a comma-separated values (csv) file into DataFrame.

Notes

See pandas API documentation for pandas.DataFrame.to_excel for more. For compatibility with `to_csv()`, to_excel serializes lists and dicts to strings before writing.

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

Examples

Create, write to and save a workbook:

```
>>> df1 = pd.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",
...              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:

```
>>> df2 = df1.copy()
>>> 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')
```

ExcelWriter can also be used to append to an existing Excel file:

```
>>> with pd.ExcelWriter('output.xlsx',
...                     mode='a') as writer:
...     df.to_excel(writer, sheet_name='Sheet_name_3')
```

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')
```

to_hdf(*path_or_buf*, *key*, *format='table'*, ***kwargs*)

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

Warning: One can store a subclass of `DataFrame` or `Series` to HDF5, but the type of the subclass is lost upon storing.

For more information see the user guide.

Parameters

- **path_or_buf** (*str or pandas.HDFStore*) – File path or HDFStore object.
- **key** (*str*) – Identifier for the group in the store.
- **mode** (*{'a', 'w', 'r+'}, default 'a'*) – Mode to open file:
  - 'w': write, a new file is created (an existing file with the same name would be deleted).
  - 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
  - 'r+': similar to 'a', but the file must already exist.
- **complevel** (*{0-9}, optional*) – Specifies a compression level for data. A value of 0 disables compression.
- **complib** (*{'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'*) – Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.
- **append** (*bool, default False*) – For Table formats, append the input data to the existing.
- **format** (*{'fixed', 'table', None}, default 'fixed'*) – Possible values:
  - 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
  - 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

  - If None, pd.get_option('io.hdf.default_format') is checked, followed by fallback to "fixed"

- **errors** (*str, default 'strict'*) – Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.
- **encoding** (*str, default "UTF-8"*) –
- **min_itemsize** (*dict or int, optional*) – Map column names to minimum string sizes for columns.
- **nan_rep** (*Any, optional*) – How to represent null values as str. Not allowed with append=True.
- **data_columns** (*list of columns or True, optional*) – List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See io.hdf5-query-data-columns. Applicable only to format='table'.

See also:

read_hdf Read from HDF file.

DataFrame.to_parquet Write a DataFrame to the binary parquet format.

DataFrame.to_sql Write to a SQL table.

DataFrame.to_feather Write out feather-format for DataFrames.

DataFrame.to_csv Write out to a csv file.

Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

Notes

See pandas API documentation for pandas.DataFrame.to_hdf for more.

to_json(*path_or_buf=None*, *orient=None*, *date_format=None*, *double_precision=10*, *force_ascii=True*, *date_unit='ms'*, *default_handler=None*, *lines=False*, *compression='infer'*, *index=True*, *indent=None*, *storage_options: Optional[Dict[str, Any]] = None*)

Convert the object to a JSON string.

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

Parameters

- **path_or_buf** (*str or file handle, optional*) – File path or object. If not specified, the result is returned as a string.
- **orient** (*str*) – Indication of expected JSON string format.
  - Series:
    - default is 'index'
    - allowed values are: {'split', 'records', 'index', 'table'}.
  - DataFrame:
    - default is 'columns'
    - allowed values are: {'split', 'records', 'index', 'columns', 'values', 'table'}.
  - The format of the JSON string:
    - 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
    - 'records' : list like [{column -> value}, … , {column -> value}]
    - 'index' : dict like {index -> {column -> value}}
    - 'columns' : dict like {column -> {index -> value}}
    - 'values' : just the values array
    - 'table' : dict like {'schema': {schema}, 'data': {data}}

      Describing the data, where data component is like `orient='records'`.
- **date_format** (*{None, 'epoch', 'iso'}*) – Type of date conversion. 'epoch' = epoch milliseconds, 'iso' = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is 'iso'. For all other orients, the default is 'epoch'.
- **double_precision** (*int, default 10*) – The number of decimal places to use when encoding floating point values.
- **force_ascii** (*bool, default True*) – Force encoded string to be ASCII.
- **date_unit** (*str, default 'ms' (milliseconds)*) – The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond, microsecond, and nanosecond respectively.

- **default_handler** (*callable, default None*) – Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.
- **lines** (*bool, default False*) – If 'orient' is 'records' write out line-delimited json format. Will throw ValueError if incorrect 'orient' since others are not list-like.
- **compression** (*{'infer', 'gzip', 'bz2', 'zip', '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.
- **index** (*bool, default True*) – Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is 'split' or 'table'.
- **indent** (*int, optional*) – Length of whitespace used to indent each record.

  New in version 1.0.0.
- **storage_options** (*dict, optional*) – Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec`. Please see `fsspec` and `urllib` for more details.

  New in version 1.2.0.

Returns If path_or_buf is None, returns the resulting json format as a string. Otherwise returns None.

Return type None or str

See also:

read_json Convert a JSON string to pandas object.

Notes

See pandas API documentation for pandas.DataFrame.to_json for more. The behavior of `indent=0` varies from the stdlib, which does not indent the output but does insert newlines. Currently, `indent=0` and the default `indent=None` are equivalent in pandas, though this may change in a future release.

`orient='table'` contains a 'pandas_version' field under 'schema'. This stores the version of *pandas* used in the latest revision of the schema.

Examples

```
>>> import json
>>> df = pd.DataFrame(
...     [["a", "b"], ["c", "d"]],
...     index=["row 1", "row 2"],
...     columns=["col 1", "col 2"],
... )
```

```
>>> result = df.to_json(orient="split")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
```

(continues on next page)

(continued from previous page)

```
{
    "columns": [
        "col 1",
        "col 2"
    ],
    "index": [
        "row 1",
        "row 2"
    ],
    "data": [
        [
            "a",
            "b"
        ],
        [
            "c",
            "d"
        ]
    ]
}
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> result = df.to_json(orient="records")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
[
    {
        "col 1": "a",
        "col 2": "b"
    },
    {
        "col 1": "c",
        "col 2": "d"
    }
]
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> result = df.to_json(orient="index")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
{
    "row 1": {
        "col 1": "a",
        "col 2": "b"
    },
    "row 2": {
        "col 1": "c",
        "col 2": "d"
    }
}
```

Encoding/decoding a Dataframe using `'columns'` formatted JSON:

```
>>> result = df.to_json(orient="columns")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
{
    "col 1": {
        "row 1": "a",
        "row 2": "c"
    },
    "col 2": {
        "row 1": "b",
        "row 2": "d"
    }
}
```

Encoding/decoding a Dataframe using `'values'` formatted JSON:

```
>>> result = df.to_json(orient="values")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
[
    [
        "a",
        "b"
    ],
    [
        "c",
        "d"
    ]
]
```

Encoding with Table Schema:

```
>>> result = df.to_json(orient="table")
>>> parsed = json.loads(result)
>>> json.dumps(parsed, indent=4)
{
    "schema": {
        "fields": [
            {
                "name": "index",
                "type": "string"
            },
            {
                "name": "col 1",
                "type": "string"
            },
            {
                "name": "col 2",
                "type": "string"
            }
        ],
        "primaryKey": [
```

(continues on next page)

(continued from previous page)

```
            "index"
        ],
        "pandas_version": "0.20.0"
    },
    "data": [
        {
            "index": "row 1",
            "col 1": "a",
            "col 2": "b"
        },
        {
            "index": "row 2",
            "col 1": "c",
            "col 2": "d"
        }
    ]
}
```

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*, *caption=None*, *label=None*, *position=None*)

Render object to a LaTeX tabular, longtable, or nested table/tabular.

Requires `\usepackage{booktabs}`. The output can be copy/pasted into a main LaTeX document or read from an external file with `\input{table.tex}`.

Changed in version 1.0.0: Added caption and label arguments.

Changed in version 1.2.0: Added position argument, changed meaning of caption argument.

Parameters

- **buf** (*str, Path or StringIO-like, optional, default 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** (*one-parameter function or str, optional, default None*) – Formatter for floating point numbers. For example `float_format="%.2f"` and `float_format="{:0.2f}".format` will both result in 0.1234 being formatted as 0.12.

- **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 'utf-8'.
- **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.
- **caption** (*str or tuple, optional*) – Tuple (full_caption, short_caption), which results in `\caption[short_caption]{full_caption}`; if a single string is passed, no short caption will be set.

  New in version 1.0.0.

  Changed in version 1.2.0: Optionally allow caption to be a tuple `(full_caption, short_caption)`.
- **label** (*str, optional*) – The LaTeX label to be placed inside `\label{}` in the output. This is used with `\ref{}` in the main `.tex` file.

  New in version 1.0.0.
- **position** (*str, optional*) – The LaTeX positional argument for tables, to be placed after `\begin{}` in the output.

  New in version 1.2.0:

  str or None If buf is None, returns the result 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 = pd.DataFrame(dict(name=['Raphael', 'Donatello'],
...                        mask=['red', 'purple'],
...                        weapon=['sai', 'bo staff']))
>>> print(df.to_latex(index=False))
\begin{tabular}{lll}
 \toprule
       name &    mask &    weapon \\
 \midrule
    Raphael &     red &       sai \\
  Donatello &  purple &  bo staff \\
\bottomrule
\end{tabular}
```

Notes

See pandas API documentation for pandas.DataFrame.to_latex for more.

to_markdown(*buf=None*, *mode: str = 'wt'*, *index: modin.pandas.base.BasePandasDataset.bool = True*, *storage_options: Optional[Dict[str, Any]] = None*, ***kwargs*)

Print DataFrame in Markdown-friendly format.

New in version 1.0.0.

Parameters

- **buf** (*str, Path or StringIO-like, optional, default None*) – Buffer to write to. If None, the output is returned as a string.
- **mode** (*str, optional*) – Mode in which file is opened, "wt" by default.
- **index** (*bool, optional, default True*) – Add index (row) labels.

  New in version 1.1.0.
- **storage_options** (*dict, optional*) – Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec`. Please see `fsspec` and `urllib` for more details.

  New in version 1.2.0.
- ****kwargs** – These parameters will be passed to tabulate.

Returns DataFrame in Markdown-friendly format.

Return type str

Notes

See pandas API documentation for pandas.DataFrame.to_markdown for more. Requires the tabulate package.

Examples

```
>>> s = pd.Series(["elk", "pig", "dog", "quetzal"], name="animal")
>>> print(s.to_markdown())
|    | animal   |
|---:|:---------|
|  0 | elk      |
|  1 | pig      |
|  2 | dog      |
|  3 | quetzal  |
```

Output markdown with a tabulate option.

```
>>> print(s.to_markdown(tablefmt="grid"))
+----+----------+
|    | animal   |
+====+==========+
|  0 | elk      |
+----+----------+
|  1 | pig      |
+----+----------+
|  2 | dog      |
+----+----------+
|  3 | quetzal  |
+----+----------+
```

to_numpy(*dtype=None*, *copy=False*, *na_value=NoDefault.no_default*)

Convert the DataFrame to a NumPy array.

By default, the dtype of the returned array will be the common NumPy dtype of all types in the DataFrame. For example, if the dtypes are `float16` and `float32`, the results dtype will be `float32`. This may require copying data and coercing values, which may be expensive.

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 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.
- **na_value** (*Any, optional*) – The value to use for missing values. The default value depends on *dtype* and the dtypes of the DataFrame columns.

  New in version 1.1.0.

Returns

Return type numpy.ndarray

See also:

Series.to_numpy Similar method for Series.

Examples

```
>>> pd.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.

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [3.0, 4.5]})
>>> df.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['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)
```

Notes

See pandas API documentation for pandas.DataFrame.to_numpy for more.

to_period(*freq=None*, *axis=0*, *copy=True*)

Convert DataFrame from DatetimeIndex to PeriodIndex.

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed).

Parameters

- **freq** (*str, default*) – Frequency of the PeriodIndex.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – The axis to convert (the index by default).
- **copy** (*bool, default True*) – If False then underlying input data is not copied.

Returns

Return type DataFrame with PeriodIndex

Notes

See pandas API documentation for pandas.DataFrame.to_period for more.

to_pickle(*path: Union[PathLike[str], str, IO, io.RawIOBase, io.BufferedIOBase, io.TextIOBase, _io.TextIOWrapper, mmap.mmap]*, *compression: Optional[Union[str, Dict[str, Any]]] = 'infer'*, *protocol: int = 4*, *storage_options: Optional[Dict[str, Any]] = None*)

Pickle (serialize) object to file.

Parameters

- **path** (*str*) – File path where the pickled object will be stored.

- **compression** (*{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'*) – A string representing the compression to use in the output file. By default, infers from the file extension in specified path. Compression mode may be any of the following possible values: {'infer', 'gzip', 'bz2', 'zip', 'xz', None}. If compression mode is 'infer' and path_or_buf is path-like, then detect compression mode from the following extensions: '.gz', '.bz2', '.zip' or '.xz'. (otherwise no compression). If dict given and mode is 'zip' or inferred as 'zip', other entries passed as additional compression options.
- **protocol** (*int*) – Int which indicates which protocol should be used by the pickler, default HIGHEST_PROTOCOL (see [1]_ paragraph 12.1.2). The possible values are 0, 1, 2, 3, 4, 5. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST_PROTOCOL.
- **storage_options** (*dict, optional*) – Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec`. Please see `fsspec` and `urllib` for more details.

  New in version 1.2.0.

See also:

read_pickle Load pickled pandas object (or any object) from file.

DataFrame.to_hdf Write DataFrame to an HDF5 file.

DataFrame.to_sql Write DataFrame to a SQL database.

DataFrame.to_parquet Write a DataFrame to the binary parquet format.

Examples

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

Notes

See pandas API documentation for pandas.DataFrame.to_pickle for more.

to_sql(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*, *method=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1]_ are supported. Tables can be newly created, appended to, or overwritten.

Parameters

- **name** (*str*) – Name of SQL table.
- **con** (*sqlalchemy.engine.(Engine or Connection) or sqlite3.Connection*) – Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects. The user is responsible for engine disposal and connection closure for the SQLAlchemy connectable See here.
- **schema** (*str, optional*) – Specify the schema (if database flavor supports this). If None, use default schema.
- **if_exists** (*{'fail', 'replace', 'append'}, default 'fail'*) – How to behave if the table already exists.
  - fail: Raise a ValueError.
  - replace: Drop the table before inserting new values.
  - append: Insert new values to the existing table.
- **index** (*bool, default True*) – Write DataFrame index as a column. Uses *index_label* as the column name in the table.
- **index_label** (*str or sequence, default None*) – Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.
- **chunksize** (*int, optional*) – Specify the number of rows in each batch to be written at a time. By default, all rows will be written at once.
- **dtype** (*dict or scalar, optional*) – Specifying the datatype for columns. If a dictionary is used, the keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode. If a scalar is provided, it will be applied to all columns.
- **method** (*{None, 'multi', callable}, optional*) – Controls the SQL insertion clause used:
  - None : Uses standard SQL `INSERT` clause (one per row).
  - 'multi': Pass multiple values in a single `INSERT` clause.
  - callable with signature `(pd_table, conn, keys, data_iter)`.

  Details and a sample callable implementation can be found in the section insert method.

Raises **ValueError** – When the table already exists and *if_exists* is 'fail' (the default).

See also:

read_sql Read a DataFrame from a table.

Notes

See pandas API documentation for pandas.DataFrame.to_sql for more. Timezone aware datetime columns will be written as `Timestamp with timezone` type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

References

Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

An *sqlalchemy.engine.Connection* can also be passed to *con*:

```
>>> with engine.begin() as connection:
...     df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
...     df1.to_sql('users', con=connection, if_exists='append')
```

This is allowed to support operations that require that the same DBAPI connection is used for the entire operation.

```
>>> df2 = pd.DataFrame({'name' : ['User 6', 'User 7']})
>>> df2.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5'), (0, 'User 6'),
 (1, 'User 7')]
```

Overwrite the table with just `df2`.

```
>>> df2.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 6'), (1, 'User 7')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

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*, *min_rows=None*, *max_cols=None*, *show_dimensions=False*, *decimal='.'*, *line_width=None*, *max_colwidth=None*, *encoding=None*)

Render a DataFrame to a console-friendly tabular output.

Parameters

- **buf** (*str, Path or StringIO-like, optional, default None*) – Buffer to write to. If None, the output is returned as a string.
- **columns** (*sequence, optional, default None*) – The subset of columns to write. Writes all columns by default.
- **col_space** (*int, list or dict of int, optional*) – The minimum width of each column.
- **header** (*bool or sequence, 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, tuple 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/tuple 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. This function must return a unicode string and will be applied only to the non-`NaN` elements, with `NaN` being handled by `na_rep`.

  Changed in version 1.2.0.
- **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.
- **min_rows** (*int, optional*) – The number of rows to display in the console in a truncated repr (when number of rows is above *max_rows*).
- **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.
- **max_colwidth** (*int, optional*) – Max width to truncate each column in characters. By default, no limit.

  New in version 1.0.0.
- **encoding** (*str, default "utf-8"*) – Set character encoding.

  New in version 1.0.

Returns If buf is None, returns the result as a string. Otherwise returns None.

Return type str or None

See also:

to_html Convert DataFrame to HTML.

Examples

```
>>> d = {'col1': [1, 2, 3], 'col2': [4, 5, 6]}
>>> df = pd.DataFrame(d)
>>> print(df.to_string())
   col1  col2
0     1     4
1     2     5
2     3     6
```

Notes

See pandas API documentation for pandas.DataFrame.to_string for more.

to_timestamp(*freq=None*, *how='start'*, *axis=0*, *copy=True*)
Cast to DatetimeIndex of timestamps, at *beginning* of period.

Parameters

- **freq** (*str, default frequency of PeriodIndex*) – Desired frequency.
- **how** (*{'s', 'e', 'start', 'end'}*) – Convention for converting period to timestamp; start of period vs. end.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – The axis to convert (the index by default).
- **copy** (*bool, default True*) – If False then underlying input data is not copied.

Returns

Return type DataFrame with DatetimeIndex

Notes

See pandas API documentation for pandas.DataFrame.to_timestamp for more.

to_xarray()
Return an xarray object from the pandas object.

Returns Data in the pandas structure converted to Dataset if the object is a DataFrame, or a DataArray if the object is a Series.

Return type xarray.DataArray or xarray.Dataset

See also:

DataFrame.to_hdf Write DataFrame to an HDF5 file.

DataFrame.to_parquet Write a DataFrame to the binary parquet format.

Notes

See pandas API documentation for pandas.DataFrame.to_xarray for more. See the xarray docs

Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0, 2),
...                    ('parrot', 'bird', 24.0, 2),
...                    ('lion', 'mammal', 80.5, 4),
...                    ('monkey', 'mammal', np.nan, 4)],
...                   columns=['name', 'class', 'max_speed',
...                            'num_legs'])
>>> df
     name   class  max_speed  num_legs
0  falcon    bird      389.0         2
1  parrot    bird       24.0         2
2    lion  mammal       80.5         4
3  monkey  mammal        NaN         4
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:    (index: 4)
Coordinates:
  * index      (index) int64 0 1 2 3
Data variables:
    name       (index) object 'falcon' 'parrot' 'lion' 'monkey'
    class      (index) object 'bird' 'bird' 'mammal' 'mammal'
    max_speed  (index) float64 389.0 24.0 80.5 nan
    num_legs   (index) int64 2 2 4 4
```

```
>>> df['max_speed'].to_xarray()
<xarray.DataArray 'max_speed' (index: 4)>
array([389. ,  24. ,  80.5,   nan])
Coordinates:
  * index    (index) int64 0 1 2 3
```

```
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-01',
...                         '2018-01-02', '2018-01-02'])
>>> df_multiindex = pd.DataFrame({'date': dates,
...                               'animal': ['falcon', 'parrot',
...                                          'falcon', 'parrot'],
...                               'speed': [350, 18, 361, 15]})
>>> df_multiindex = df_multiindex.set_index(['date', 'animal'])
```

```
>>> df_multiindex
                   speed
date       animal
2018-01-01 falcon    350
           parrot     18
2018-01-02 falcon    361
           parrot     15
```

```
>>> df_multiindex.to_xarray()
<xarray.Dataset>
Dimensions:  (animal: 2, date: 2)
Coordinates:
  * date     (date) datetime64[ns] 2018-01-01 2018-01-02
  * animal   (animal) object 'falcon' 'parrot'
Data variables:
    speed    (date, animal) int64 350 18 361 15
```

transform(*func*, *axis=0*, **args*, ***kwargs*)

Call `func` on self producing a DataFrame with transformed values.

Produced DataFrame will have same axis length as self.

Parameters

- **func** (*function, str, list-like or dict-like*) – Function to use for transforming the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. If func is both list-like and dict-like, dict-like behavior takes precedence.

  Accepted combinations are:

  - function
  - string function name
  - list-like of functions and/or function names, e.g. `[np.exp, 'sqrt']`
  - dict-like of axis labels -> functions, function names or list-like of such.

- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – If 0 or 'index': apply function to each column. If 1 or 'columns': apply function to each row.
- ***args** – Positional arguments to pass to *func*.
- ****kwargs** – Keyword arguments to pass to *func*.

Returns A DataFrame that must have the same length as self.

Return type DataFrame

:raises ValueError : If the returned DataFrame has a different length than self.:

See also:

DataFrame.agg Only perform aggregating type operations.

DataFrame.apply Invoke function on a DataFrame.

Notes

See pandas API documentation for pandas.DataFrame.transform for more. Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See gotchas.udf-mutation for more details.

Examples

```
>>> df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})
>>> df
   A  B
0  0  1
1  1  2
2  2  3
>>> df.transform(lambda x: x + 1)
   A  B
0  1  2
1  2  3
2  3  4
```

Even though the resulting DataFrame must have the same length as the input DataFrame, it is possible to provide several input functions:

```
>>> s = pd.Series(range(3))
>>> s
0    0
1    1
2    2
dtype: int64
>>> s.transform([np.sqrt, np.exp])
       sqrt        exp
0  0.000000   1.000000
1  1.000000   2.718282
2  1.414214   7.389056
```

You can call transform on a GroupBy object:

```
>>> df = pd.DataFrame({
...     "Date": [
...         "2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05",
...         "2015-05-08", "2015-05-07", "2015-05-06", "2015-05-05"],
...     "Data": [5, 8, 6, 1, 50, 100, 60, 120],
... })
>>> df
         Date  Data
0  2015-05-08     5
1  2015-05-07     8
2  2015-05-06     6
3  2015-05-05     1
4  2015-05-08    50
5  2015-05-07   100
6  2015-05-06    60
7  2015-05-05   120
>>> df.groupby('Date')['Data'].transform('sum')
0     55
1    108
2     66
3    121
4     55
```

(continues on next page)

(continued from previous page)

```
5    108
6     66
7    121
Name: Data, dtype: int64
```

```
>>> df = pd.DataFrame({
...     "c": [1, 1, 1, 2, 2, 2, 2],
...     "type": ["m", "n", "o", "m", "m", "n", "n"]
... })
>>> df
   c type
0  1    m
1  1    n
2  1    o
3  2    m
4  2    m
5  2    n
6  2    n
>>> df['size'] = df.groupby('c')['type'].transform(len)
>>> df
   c type size
0  1    m    3
1  1    n    3
2  1    o    3
3  2    m    4
4  2    m    4
5  2    n    4
6  2    n    4
```

truediv(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Get Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs. With reverse version, *rtruediv*.

Among flexible wrappers (*add*, *sub*, *mul*, *div*, *mod*, *pow*) to arithmetic operators: +, -, *, /, //, %, **.

Parameters

- **other** (*scalar, sequence, Series, or DataFrame*) – Any single or multiple element data structure, or list-like object.
- **axis** (*{0 or 'index', 1 or 'columns'}*) – Whether to compare by the index (0 or 'index') or columns (1 or 'columns'). For Series input, axis to match Series index on.
- **level** (*int or label*) – Broadcast across a level, matching Index values on the passed MultiIndex level.
- **fill_value** (*float or None, default None*) – Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing.

Returns Result of the arithmetic operation.

Return type DataFrame

See also:

DataFrame.add Add DataFrames.

DataFrame.sub Subtract DataFrames.

DataFrame.mul Multiply DataFrames.

DataFrame.div Divide DataFrames (float division).

DataFrame.truediv Divide DataFrames (float division).

DataFrame.floordiv Divide DataFrames (integer division).

DataFrame.mod Calculate modulo (remainder after division).

DataFrame.pow Calculate exponential power.

Notes

See pandas API documentation for pandas.DataFrame.truediv for more. Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> 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
```

Divide by constant with reverse version.

```
>>> 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
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2], axis='columns')
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub(pd.Series([1, 1, 1], index=['circle', 'triangle', 'rectangle']),
...         axis='index')
           angles  degrees
circle         -1      359
triangle        2      179
rectangle       3      359
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({'angles': [0, 3, 4]},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other, fill_value=0)
           angles  degrees
circle          0      0.0
triangle        9      0.0
rectangle      16      0.0
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

```
>>> df.div(df_multindex, level=1, fill_value=0)
             angles  degrees
A circle        NaN      1.0
  triangle      1.0      1.0
  rectangle     1.0      1.0
B square        0.0      0.0
  pentagon      0.0      0.0
  hexagon       0.0      0.0
```

truncate(*before=None*, *after=None*, *axis=None*, *copy=True*)

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.

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 The truncated Series or DataFrame.

Return type type of caller

See also:

DataFrame.loc Select a subset of a DataFrame by label.

DataFrame.iloc Select a subset of a DataFrame by position.

Notes

See pandas API documentation for pandas.DataFrame.truncate for more. If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

Examples

```
>>> df = pd.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
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                     A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
```

(continues on next page)

(continued from previous page)

```
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...              after=pd.Timestamp('2016-01-10')).tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a DatetimeIndex containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                     A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                     A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
2016-01-10 23:59:58  1
2016-01-10 23:59:59  1
```

tshift(*periods=1*, *freq=None*, *axis=0*)

Shift the time index, using the index's frequency if available.

Deprecated since version 1.1.0: Use *shift* instead.

Parameters

- **periods** (*int*) – Number of periods to move, can be positive or negative.
- **freq** (*DateOffset, timedelta, or str, default None*) – Increment to use from the tseries module or time rule expressed as a string (e.g. 'EOM').
- **axis** (*{0 or 'index', 1 or 'columns', None}, default 0*) – Corresponds to the axis that contains the Index.

Returns **shifted**

Return type Series/DataFrame

Notes

See pandas API documentation for pandas.DataFrame.tshift for more. If freq is not specified then tries to use the freq or inferred_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

tz_convert(*tz*, *axis=0*, *level=None*, *copy=True*)
Convert tz-aware axis to target time zone.

Parameters

- **tz** (*str or tzinfo object*) –
- **axis** (*the axis to convert*) –
- **level** (*int, str, default None*) – If axis is a MultiIndex, convert a specific level. Otherwise must be None.
- **copy** (*bool, default True*) – Also make a copy of the underlying data.

Returns Object with time zone converted axis.

Return type {klass}

Raises **TypeError** – If the axis is tz-naive.

Notes

See pandas API documentation for pandas.DataFrame.tz_convert for more.

tz_localize(*tz*, *axis=0*, *level=None*, *copy=True*, *ambiguous='raise'*, *nonexistent='raise'*)
Localize tz-naive index of a Series or DataFrame to target time zone.

This operation localizes the Index. To localize the values in a timezone-naive Series, use `Series.dt.tz_localize()`.

Parameters

- **tz** (*str or tzinfo*) –
- **axis** (*the axis to localize*) –
- **level** (*int, str, default None*) – If axis ia a MultiIndex, localize a specific level. Otherwise must be None.
- **copy** (*bool, default True*) – Also make a copy of the underlying data.
- **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 designates 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.

- **nonexistent** (*str, default 'raise'*) – A nonexistent time does not exist in a particular timezone where clocks moved forward due to DST. Valid values are:
  - '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.

Returns Same type as the input.

Return type *Series* or DataFrame

Raises **TypeError** – If the TimeSeries is tz-aware and tz is not None.

Examples

Localize local times:

```
>>> s = pd.Series([1],
...               index=pd.DatetimeIndex(['2018-09-15 01:30:00']))
>>> s.tz_localize('CET')
2018-09-15 01:30:00+02:00    1
dtype: int64
```

Be careful with DST changes. When there is sequential data, pandas can infer the DST time:

```
>>> s = pd.Series(range(7),
...               index=pd.DatetimeIndex(['2018-10-28 01:30:00',
...                                       '2018-10-28 02:00:00',
...                                       '2018-10-28 02:30:00',
...                                       '2018-10-28 02:00:00',
...                                       '2018-10-28 02:30:00',
...                                       '2018-10-28 03:00:00',
...                                       '2018-10-28 03:30:00']))
>>> s.tz_localize('CET', ambiguous='infer')
2018-10-28 01:30:00+02:00    0
2018-10-28 02:00:00+02:00    1
2018-10-28 02:30:00+02:00    2
2018-10-28 02:00:00+01:00    3
2018-10-28 02:30:00+01:00    4
2018-10-28 03:00:00+01:00    5
2018-10-28 03:30:00+01:00    6
dtype: int64
```

In some cases, inferring the DST is impossible. In such cases, you can pass an ndarray to the ambiguous parameter to set the DST explicitly

```
>>> s = pd.Series(range(3),
...               index=pd.DatetimeIndex(['2018-10-28 01:20:00',
...                                       '2018-10-28 02:36:00',
```

(continues on next page)

(continued from previous page)

```
...                                          '2018-10-28 03:46:00']))
>>> s.tz_localize('CET', ambiguous=np.array([True, True, False]))
2018-10-28 01:20:00+02:00    0
2018-10-28 02:36:00+02:00    1
2018-10-28 03:46:00+01:00    2
dtype: int64
```

If the DST transition causes nonexistent times, you can shift these dates forward or backward with a timedelta object or *'shift_forward'* or *'shift_backward'*.

```
>>> s = pd.Series(range(2),
...               index=pd.DatetimeIndex(['2015-03-29 02:30:00',
...                                       '2015-03-29 03:30:00']))
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_forward')
2015-03-29 03:00:00+02:00    0
2015-03-29 03:30:00+02:00    1
dtype: int64
>>> s.tz_localize('Europe/Warsaw', nonexistent='shift_backward')
2015-03-29 01:59:59.999999999+01:00    0
2015-03-29 03:30:00+02:00              1
dtype: int64
>>> s.tz_localize('Europe/Warsaw', nonexistent=pd.Timedelta('1H'))
2015-03-29 03:30:00+02:00    0
2015-03-29 03:30:00+02:00    1
dtype: int64
```

Notes

See pandas API documentation for pandas.DataFrame.tz_localize for more.

value_counts(*subset: Optional[Sequence[Hashable]] = None*, *normalize: modin.pandas.base.BasePandasDataset.bool = False*, *sort: modin.pandas.base.BasePandasDataset.bool = True*, *ascending: modin.pandas.base.BasePandasDataset.bool = False*, *dropna: modin.pandas.base.BasePandasDataset.bool = True*)

Return a Series containing counts of unique rows in the DataFrame.

New in version 1.1.0.

Parameters

- **subset** (*list-like, optional*) – Columns to use when counting unique combinations.
- **normalize** (*bool, default False*) – Return proportions rather than frequencies.
- **sort** (*bool, default True*) – Sort by frequencies.
- **ascending** (*bool, default False*) – Sort in ascending order.
- **dropna** (*bool, default True*) – Don't include counts of rows that contain NA values.

  New in version 1.3.0.

Returns

Return type *Series*

See also:

Series.value_counts Equivalent method on Series.

Notes

See pandas API documentation for pandas.DataFrame.value_counts for more. The returned Series will have a MultiIndex with one level per input column. By default, rows that contain any NA values are omitted from the result. By default, the resulting Series will be in descending order so that the first element is the most frequently-occurring row.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 4, 6],
...                    'num_wings': [2, 0, 0, 0]},
...                   index=['falcon', 'dog', 'cat', 'ant'])
>>> df
        num_legs  num_wings
falcon         2          2
dog            4          0
cat            4          0
ant            6          0
```

```
>>> df.value_counts()
num_legs  num_wings
4         0            2
2         2            1
6         0            1
dtype: int64
```

```
>>> df.value_counts(sort=False)
num_legs  num_wings
2         2            1
4         0            2
6         0            1
dtype: int64
```

```
>>> df.value_counts(ascending=True)
num_legs  num_wings
2         2            1
6         0            1
4         0            2
dtype: int64
```

```
>>> df.value_counts(normalize=True)
num_legs  num_wings
4         0            0.50
2         2            0.25
```

(continues on next page)

(continued from previous page)

```
6         0            0.25
dtype: float64
```

With *dropna* set to *False* we can also count rows with NA values.

```
>>> df = pd.DataFrame({'first_name': ['John', 'Anne', 'John', 'Beth'],
...                    'middle_name': ['Smith', pd.NA, pd.NA, 'Louise']})
>>> df
  first_name middle_name
0       John       Smith
1       Anne        <NA>
2       John        <NA>
3       Beth      Louise
```

```
>>> df.value_counts()
first_name  middle_name
Beth        Louise         1
John        Smith          1
dtype: int64
```

```
>>> df.value_counts(dropna=False)
first_name  middle_name
Anne        NaN            1
Beth        Louise         1
John        Smith          1
            NaN            1
dtype: int64
```

property values

Return a Numpy representation of the DataFrame.

> **Warning:** We recommend using `DataFrame.to_numpy()` instead.

Only the values in the DataFrame will be returned, the axes labels will be removed.

Returns The values of the DataFrame.

Return type numpy.ndarray

See also:

DataFrame.to_numpy Recommended alternative to this method.

DataFrame.index Retrieve the index labels.

DataFrame.columns Retrieving the column names.

Notes

See pandas API documentation for pandas.DataFrame.values for more. The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

Examples

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.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 = pd.DataFrame([('parrot',   24.0, 'second'),
...                     ('lion',     80.5, 1),
...                     ('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, 1],
       ['monkey', nan, None]], dtype=object)
```

var(*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric_only=None*, ***kwargs*)

Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

Parameters

- **axis** (*{index (0), columns (1)}*) –

- **skipna** (*bool, default True*) – Exclude NA/null values. If an entire row/column is NA, the result will be NA.
- **level** (*int or level name, default None*) – If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.
- **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. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

Return type *Series* or DataFrame (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.var for more. To have the same behaviour as *numpy.std*, use *ddof=0* (instead of the default *ddof=1*)

DataFrame Module Overview

Modin's `pandas.DataFrame` API

Modin's `pandas.DataFrame` API is backed by a distributed object providing an identical API to pandas. After the user calls some `DataFrame` function, this call is internally rewritten into a representation that can be processed in parallel by the partitions. These results can be e.g., reduced to single output, identical to the single threaded pandas `DataFrame` method output.

Public API

class modin.pandas.dataframe.**DataFrame**(*data=None*, *index=None*, *columns=None*, *dtype=None*, *copy=None*, *query_compiler=None*)

Modin distributed representation of `pandas.DataFrame`.

Internally, the data can be divided into partitions along both columns and rows in order to parallelize computations and utilize the user's hardware as much as possible.

Inherit common for `DataFrame`-s and `Series` functionality from the *BasePandasDataset* class.

Parameters

- **data** (`DataFrame`, `Series`, *pandas.DataFrame, ndarray, Iterable or dict, optional*) – Dict can contain `Series`, arrays, constants, dataclass or list-like objects. If data is a dict, column order follows insertion-order.
- **index** (*Index or array-like, optional*) – 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, optional*) – Column labels to use for resulting frame. Will default to `RangeIndex` if no column labels are provided.
- **dtype** (*str, np.dtype, or pandas.ExtensionDtype, optional*) – Data type to force. Only a single dtype is allowed. If None, infer.

- **copy** (*bool, default: False*) – Copy data from inputs. Only affects `pandas.DataFrame` / 2d ndarray input.
- **query_compiler** (`BaseQueryCompiler`, *optional*) – A query compiler object to create the `DataFrame` from.

Notes

See pandas API documentation for pandas.DataFrame for more. `DataFrame` can be created either from passed *data* or *query_compiler*. If both parameters are provided, data source will be prioritized in the next order:

1) Modin `DataFrame` or `Series` passed with *data* parameter.
2) Query compiler from the *query_compiler* parameter.
3) Various pandas/NumPy/Python data structures passed with *data* parameter.

The last option is less desirable since import of such data structures is very inefficient, please use previously created Modin structures from the fist two options or import data using highly efficient Modin IO tools (for example `pd.read_csv`).

Usage Guide

The most efficient way to create Modin `DataFrame` is to import data from external storage using the highly efficient Modin IO methods (for example using `pd.read_csv`, see details for Modin IO methods in the separate section), but even if the data does not originate from a file, any pandas supported data type or `pandas.DataFrame` can be used. Internally, the `DataFrame` data is divided into partitions, which number along an axis usually corresponds to the number of the user's hardware CPUs. If needed, the number of partitions can be changed by setting `modin.config.NPartitions`.

Let's consider simple example of creation and interacting with Modin `DataFrame`:

```
import modin.config

# This explicitly sets the number of partitions
modin.config.NPartitions.put(4)

import modin.pandas as pd
import pandas

# Create Modin DataFrame from the external file
pd_dataframe = pd.read_csv("test_data.csv")
# Create Modin DataFrame from the python object
# data = {f'col{x}': [f'col{x}_{y}' for y in range(100, 356)] for x in range(4)}
# pd_dataframe = pd.DataFrame(data)
# Create Modin DataFrame from the pandas object
# pd_dataframe = pd.DataFrame(pandas.DataFrame(data))

# Show created DataFrame
print(pd_dataframe)

# List DataFrame partitions. Note, that internal API is intended for
# developers needs and was used here for presentation purposes
# only.
partitions = pd_dataframe._query_compiler._modin_frame._partitions
print(partitions)
```

(continues on next page)

(continued from previous page)

```
# Show the first DataFrame partition
print(partitions[0][0].get())

Output:

# created DataFrame

        col0      col1      col2      col3
0    col0_100  col1_100  col2_100  col3_100
1    col0_101  col1_101  col2_101  col3_101
2    col0_102  col1_102  col2_102  col3_102
3    col0_103  col1_103  col2_103  col3_103
4    col0_104  col1_104  col2_104  col3_104
..        ...       ...       ...       ...
251  col0_351  col1_351  col2_351  col3_351
252  col0_352  col1_352  col2_352  col3_352
253  col0_353  col1_353  col2_353  col3_353
254  col0_354  col1_354  col2_354  col3_354
255  col0_355  col1_355  col2_355  col3_355

[256 rows x 4 columns]

# List of DataFrame partitions

[[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
↪PandasOnRayDataframePartition object at 0x7fc554e607f0>]
[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
↪PandasOnRayDataframePartition object at 0x7fc554e9a4f0>]
[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
↪PandasOnRayDataframePartition object at 0x7fc554e60820>]
[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
↪PandasOnRayDataframePartition object at 0x7fc554e609d0>]]

# The first DataFrame partition

        col0      col1      col2      col3
0   col0_100  col1_100  col2_100  col3_100
1   col0_101  col1_101  col2_101  col3_101
2   col0_102  col1_102  col2_102  col3_102
3   col0_103  col1_103  col2_103  col3_103
4   col0_104  col1_104  col2_104  col3_104
..       ...       ...       ...       ...
60  col0_160  col1_160  col2_160  col3_160
61  col0_161  col1_161  col2_161  col3_161
62  col0_162  col1_162  col2_162  col3_162
63  col0_163  col1_163  col2_163  col3_163
64  col0_164  col1_164  col2_164  col3_164

[65 rows x 4 columns]
```

As we show in the example above, Modin `DataFrame` can be easily created, and supports any input that pandas `DataFrame` supports. Also note that tuning of the `DataFrame` partitioning can be done by just setting a single config.

Series Module Overview

Modin's pandas.Series API

Modin's pandas.Series API is backed by a distributed object providing an identical API to pandas. After the user calls some Series function, this call is internally rewritten into a representation that can be processed in parallel by the partitions. These results can be e.g., reduced to single output, identical to the single threaded pandas Series method output.

Public API

class modin.pandas.series.**Series**(*data=None*, *index=None*, *dtype=None*, *name=None*, *copy=False*, *fastpath=False*, *query_compiler=None*)

Modin distributed representation of *pandas.Series*.

Internally, the data can be divided into partitions in order to parallelize computations and utilize the user's hardware as much as possible.

Inherit common for DataFrames and Series functionality from the *BasePandasDataset* class.

Parameters

- **data** (*modin.pandas.Series, array-like, Iterable, dict, or scalar value, optional*) – Contains data stored in Series. If data is a dict, argument order is maintained.
- **index** (*array-like or Index (1d), optional*) – Values must be hashable and have the same length as *data*.
- **dtype** (*str, np.dtype, or pandas.ExtensionDtype, optional*) – Data type for the output Series. If not specified, this will be inferred from *data*.
- **name** (*str, optional*) – The name to give to the Series.
- **copy** (*bool, default: False*) – Copy input data.
- **fastpath** (*bool, default: False*) – *pandas* internal parameter.
- **query_compiler** (*BaseQueryCompiler, optional*) – A query compiler object to create the Series from.

Notes

See pandas API documentation for pandas.Series for more.

Usage Guide

The most efficient way to create Modin Series is to import data from external storage using the highly efficient Modin IO methods (for example using pd.read_csv, see details for Modin IO methods in the separate section), but even if the data does not originate from a file, any pandas supported data type or pandas.Series can be used. Internally, the Series data is divided into partitions, which number along an axis usually corresponds to the number of the user's hardware CPUs. If needed, the number of partitions can be changed by setting modin.config.NPartitions.

Let's consider simple example of creation and interacting with Modin Series:

```
import modin.config

# This explicitly sets the number of partitions
modin.config.NPartitions.put(4)

import modin.pandas as pd
import pandas

# Create Modin Series from the external file
pd_series = pd.read_csv("test_data.csv", header=None).squeeze()
# Create Modin Series from the python object
# pd_series = pd.Series([x for x in range(256)])
# Create Modin Series from the pandas object
# pd_series = pd.Series(pandas.Series([x for x in range(256)]))

# Show created `Series`
print(pd_series)

# List `Series` partitions. Note, that internal API is intended for
# developers needs and was used here for presentation purposes
# only.
partitions = pd_series._query_compiler._modin_frame._partitions
print(partitions)

# Show the first `Series` partition
print(partitions[0][0].get())

Output:

# created `Series`

0      100
1      101
2      102
3      103
4      104
    ...
251    351
252    352
253    353
254    354
255    355
Name: 0, Length: 256, dtype: int64

# List of `Series` partitions

[[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
 ↪PandasOnRayDataframePartition object at 0x7fc554e607f0>]
[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
 ↪PandasOnRayDataframePartition object at 0x7fc554e9a4f0>]
[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
 ↪PandasOnRayDataframePartition object at 0x7fc554e60820>]
[<modin.core.execution.ray.implementations.pandas_on_ray.partitioning.partition.
 ↪PandasOnRayDataframePartition object at 0x7fc554e609d0>]]
```

(continues on next page)

(continued from previous page)

```
# The first `Series` partition

    0
0   100
1   101
2   102
3   103
4   104
..  ...
60  160
61  161
62  162
63  163
64  164

[65 rows x 1 columns]
```

As we show in the example above, Modin `Series` can be easily created, and supports any input that pandas `Series` supports. Also note that tuning of the `Series` partitioning can be done by just setting a single config.

Query Compiler

The Query Compiler receives queries from the pandas API layer. The API layer's responsibility is to ensure clean input to the Query Compiler. The Query Compiler must have knowledge of the compute kernels/in-memory format of the data in order to efficiently compile the queries.

The Query Compiler is responsible for sending the compiled query to the Core Modin Dataframe. In this design, the Query Compiler does not have information about where or when the query will be executed, and gives the control of the partition layout to the Modin Dataframe.

In the interest of reducing the pandas API, the Query Compiler layer closely follows the pandas API, but cuts out a large majority of the repetition.

Core Modin Dataframe

At this layer, operations can be performed lazily. Currently, Modin executes most operations eagerly in an attempt to behave as pandas does. Some operations, e.g. `transpose` are expensive and create full copies of the data in-memory. In these cases, we can wait until another operation triggers computation. In the future, we plan to add additional query planning and laziness to Modin to ensure that queries are performed efficiently.

The structure of the Core Modin Dataframe is extensible, such that any operation that could be better optimized for a given execution can be overridden and optimized in that way.

This layer has a significantly reduced API from the QueryCompiler and the user-facing API. Each of these APIs represents a single way of performing a given operation or behavior. Some of these are expanded for convenience/understanding. The API abstractions are as follows:

Core Modin Dataframe API

- `mask`: Indexing/masking/selecting on the data (by label or by integer index).
- `copy`: Create a copy of the data.
- `mapreduce`: Reduce the dimension of the data.
- `foldreduce`: Reduce the dimension of the data, but entire column/row information is needed.
- `map`: Perform a map.
- `fold`: Perform a fold.
- `apply_<type>`: Apply a function that may or may not change the shape of the data.
  - `full_axis`: Apply a function requires knowledge of the entire axis.
  - `full_axis_select_indices`: Apply a function performed on a subset of the data that requires knowledge of the entire axis.
  - `select_indices`: Apply a function to a subset of the data. This is mainly used for indexing.
- `binary_op`: Perform a function between two dataframes.
- `concat`: Append one or more dataframes to either axis of this dataframe.
- `transpose`: Swap the axes (columns become rows, rows become columns).
- `groupby`:
  - `groupby_reduce`: Perform a reduction on each group.
  - `groupby_apply`: Apply a function to each group.
- **take functions**
  - `head`: Take the first `n` rows.
  - `tail`: Take the last `n` rows.
  - `front`: Take the first `n` columns.
  - `back`: Take the last `n` columns.
- **import/export functions**
  - `from_pandas`: Convert a pandas dataframe to a Modin dataframe.
  - `to_pandas`: Convert a Modin dataframe to a pandas dataframe.
  - `to_numpy`: Convert a Modin dataframe to a numpy array.

More documentation can be found internally in the code. This API is not complete, but represents an overwhelming majority of operations and behaviors.

This API can be implemented by other distributed/parallel DataFrame libraries and plugged in to Modin as well. Create an issue or discuss on our Discourse for more information!

The Core Modin Dataframe is responsible for the data layout and shuffling, partitioning, and serializing the tasks that get sent to each partition. Other implementations of the Modin Dataframe interface will have to handle these as well.

Execution Engine/Framework

This layer is what Modin uses to perform computation on a partition of the data. The Core Modin Dataframe is designed to work with task parallel frameworks, but with some effort, a data parallel framework is possible.

Internal abstractions

These abstractions are not included in the above architecture, but are important to the internals of Modin.

Partition Manager

The Partition Manager can change the size and shape of the partitions based on the type of operation. For example, certain operations are complex and require access to an entire column or row. The Partition Manager can convert the block partitions to row partitions or column partitions. This gives Modin the flexibility to perform operations that are difficult in row-only or column-only partitioning schemas.

Another important component of the Partition Manager is the serialization and shipment of compiled queries to the Partitions. It maintains metadata for the length and width of each partition, so when operations only need to operate on or extract a subset of the data, it can ship those queries directly to the correct partition. This is particularly important for some operations in pandas which can accept different arguments and operations for different columns, e.g. `fillna` with a dictionary.

This abstraction separates the actual data movement and function application from the Dataframe layer to keep the Core Dataframe API small and separately optimize the data movement and metadata management.

Partition

Partitions are responsible for managing a subset of the Dataframe. As is mentioned above, the Dataframe is partitioned both row and column-wise. This gives Modin scalability in both directions and flexibility in data layout. There are a number of optimizations in Modin that are implemented in the partitions. Partitions are specific to the execution framework and in-memory format of the data. This allows Modin to exploit potential optimizations across both of these. These optimizations are explained further on the pages specific to the execution framework.

Supported Execution Frameworks and Memory Formats

This is the list of execution frameworks and memory formats supported in Modin. If you would like to contribute a new execution framework or memory format, please see the documentation page on *contributing*.

- ***Pandas on Ray***
  - Uses the Ray execution framework.
  - The compute kernel/in-memory format is a pandas DataFrame.
- ***Pandas on Dask***
  - Uses the Dask Futures execution framework.
  - The compute kernel/in-memory format is a pandas DataFrame.
- ***Omnisci***
  - Uses OmniSciDB as an engine.
  - The compute kernel/in-memory format is a pyarrow Table or pandas DataFrame when defaulting to pandas.

- *Pyarrow on Ray* **(experimental)**
  - Uses the Ray execution framework.
  - The compute kernel/in-memory format is a pyarrow Table.

3.20.6 Module/Class View

Modin modules layout is shown below. To deep dive into Modin internal implementation details just pick module you are interested in (only some of the modules are covered by documentation for now, the rest is coming soon...).

```
├──.github
├──asv_bench
├──ci
├──docker
├──docs
├──examples
├──modin
    ├── config
    ├──core
        ├── dataframe
            ├── algebra
            ├── base
            └── pandas
        ├──execution
            ├──dask
                ├──common
                └──implementations
                    └── pandas_on_dask
            ├── dispatching
            ├──python
                └──implementations
                    └── pandas_on_python
            └──ray
                ├──common
                ├── generic
                └──implementations
                    ├── cudf_on_ray
                    └── pandas_on_ray
        ├── io
        └── storage_formats
            ├── base
            ├──cudf
            ├── pandas
            └── pyarrow
    ├──distributed
        ├──dataframe
            └── pandas
    ├── experimental
        ├──cloud
        ├──core
            ├──execution
                ├──native
                    └──implementations
                        └── omnisci_on_native
```

```
        └──ray
            └──implementations
                ├── pandas_on_ray
                └── pyarrow_on_ray
    └── storage_formats
        └── omnisci
├── pandas
├── sklearn
├──spreadsheet
├──sql
└── xgboost
└──pandas
    ├── dataframe
    └── series
├──requirements
├──scripts
└──stress_tests
```

3.21 Partition API in Modin

When you are working with a *DataFrame*, you can unwrap its remote partitions to get the raw futures objects compatible with the execution engine (e.g. `ray.ObjectRef` for Ray). In addition to unwrapping of the remote partitions we also provide an API to construct a `modin.pandas.DataFrame` from raw futures objects.

3.21.1 Partition IPs

For finer grained placement control, Modin also provides an API to get the IP addresses of the nodes that hold each partition. You can pass the partitions having needed IPs to your function. It can help with minimazing of data movement between nodes.

3.21.2 Partition API implementations

By default, a *DataFrame* stores underlying partitions as `pandas.DataFrame` objects. You can find the specific implementation of Modin's Partition Interface in *Pandas Partition API*.

Pandas Partition API

This page contains a description of the API to extract partitions from and build Modin Dataframes.

unwrap_partitions

`modin.distributed.dataframe.pandas.`**`unwrap_partitions`**(*api_layer_object*, *axis=None*, *get_ip=False*)

Unwrap partitions of the `api_layer_object`.

Parameters

- **`api_layer_object`** (*`DataFrame or Series`*) – The API layer object.
- **`axis`** (*`{None, 0, 1}, default: None`*) – The axis to unwrap partitions for (0 - row partitions, 1 - column partitions). If `axis is None`, the partitions are unwrapped as they are currently stored.

- **get_ip** (*bool, default: False*) – Whether to get node ip address to each partition or not.

Returns A list of Ray.ObjectRef/Dask.Future to partitions of the `api_layer_object` if Ray/Dask is used as an engine.

Return type list

Notes

If `get_ip=True`, a list of tuples of Ray.ObjectRef/Dask.Future to node ip addresses and partitions of the `api_layer_object`, respectively, is returned if Ray/Dask is used as an engine (i.e. `[(Ray.ObjectRef/Dask.Future, Ray.ObjectRef/Dask.Future), ...]`).

from_partitions

`modin.distributed.dataframe.pandas.`**from_partitions**(*partitions*, *axis*, *index=None*, *columns=None*, *row_lengths=None*, *column_widths=None*)

Create DataFrame from remote partitions.

Parameters

- **partitions** (*list*) – A list of Ray.ObjectRef/Dask.Future to partitions depending on the engine used. Or a list of tuples of Ray.ObjectRef/Dask.Future to node ip addresses and partitions depending on the engine used (i.e. `[(Ray.ObjectRef/Dask.Future, Ray.ObjectRef/Dask.Future), ...]`).
- **axis** (*{None, 0 or 1}*) – The `axis` parameter is used to identify what are the partitions passed. You have to set:
  - `axis=0` if you want to create DataFrame from row partitions
  - `axis=1` if you want to create DataFrame from column partitions
  - `axis=None` if you want to create DataFrame from 2D list of partitions
- **index** (*sequence, optional*) – The index for the DataFrame. Is computed if not provided.
- **columns** (*sequence, optional*) – The columns for the DataFrame. Is computed if not provided.
- **row_lengths** (*list, optional*) – The length of each partition in the rows. The "height" of each of the block partitions. Is computed if not provided.
- **column_widths** (*list, optional*) – The width of each partition in the columns. The "width" of each of the block partitions. Is computed if not provided.

Returns DataFrame instance created from remote partitions.

Return type modin.pandas.DataFrame

Notes

Pass *index*, *columns*, *row_lengths* and *column_widths* to avoid triggering extra computations of the metadata when creating a DataFrame.

Example

```
import modin.pandas as pd
from modin.distributed.dataframe.pandas import unwrap_partitions, from_partitions
import numpy as np
data = np.random.randint(0, 100, size=(2 ** 10, 2 ** 8))
df = pd.DataFrame(data)
partitions = unwrap_partitions(df, axis=0, get_ip=True)
print(partitions)
new_df = from_partitions(partitions, axis=0)
print(new_df)
```

3.21.3 Ray engine

However, it is worth noting that for Modin on `Ray` engine with `pandas` in-memory format IPs of the remote partitions may not match actual locations if the partitions are lower than 100 kB. Ray saves such objects (<= 100 kB, by default) in in-process store of the calling process (please, refer to Ray documentation for more information). We can't get IPs for such objects while maintaining good performance. So, you should keep in mind this for unwrapping of the remote partitions with their IPs. Several options are provided to handle the case in `How to handle Ray objects that are lower 100 kB` section.

3.21.4 Dask engine

There is no mentioned above issue for Modin on `Dask` engine with `pandas` in-memory format because `Dask` saves any objects in the worker process that processes a function (please, refer to Dask documentation for more information).

3.21.5 How to handle Ray objects that are lower than 100 kB

- If you are sure that each of the remote partitions being unwrapped is higher than 100 kB, you can just import Modin or perform `ray.init()` manually.
- If you don't know partition sizes you can pass the option `_system_config={"max_direct_call_object_size": <nbytes>,}`, where `nbytes` is threshold for objects that will be stored in in-process store, to `ray.init()`.
- You can also start Ray as follows: `ray start --head --system-config='{"max_direct_call_object_size":<nbytes>`

Note that when specifying the threshold the performance of some Modin operations may change.

3.22 pandas on Ray

This section describes usage related documents for the pandas on Ray component of Modin.

Modin uses pandas as a primary memory format of the underlying partitions and optimizes queries ingested from the API layer in a specific way to this format. Thus, there is no need to care of choosing it but you can explicitly specify it anyway as shown below.

One of the execution engines that Modin uses is Ray. If you have Ray installed in your system, Modin also uses it by default to distribute computations.

If you want to be explicit, you could set the following environment variables:

```
export MODIN_ENGINE=ray
export MODIN_STORAGE_FORMAT=pandas
```

or turn it on in source code:

```
import modin.config as cfg
cfg.Engine.put('ray')
cfg.StorageFormat.put('pandas')
```

3.23 Pandas on Dask

The Dask engine and documentation could use your help! Consider opening a pull request or an issue to contribute or ask clarifying questions.

3.24 OmniSci

This section describes usage related documents for the OmniSciDB-based engine of Modin.

This engine uses analytical database OmniSciDB to obtain high single-node scalability for specific set of dataframe operations. To enable this engine you could set the following environment variables:

```
export MODIN_ENGINE=native
export MODIN_STORAGE_FORMAT=omnisci
export MODIN_EXPERIMENTAL=true
```

or turn it on in source code:

```
import modin.config as cfg
cfg.Engine.put('native')
cfg.StorageFormat.put('omnisci')
cfg.IsExperimental.put(True)
```

To enable `OmniSci` engine launch export :

```
export MODIN_STORAGE_FORMAT=omnisci
```

or use it in your code:

```
import modin.config as cfg
cfg.StorageFormat.put('omnisci')
```

3.25 Pyarrow on Ray

Coming Soon!

3.26 Troubleshooting

We hope your experience with Modin is bug-free, but there are some quirks about Modin that may require troubleshooting.

3.26.1 Frequently encountered issues

This is a list of the most frequently encountered issues when using Modin. Some of these are working as intended, while others are known bugs that are being actively worked on.

Error During execution: `ArrowIOError: Broken Pipe`

One of the more frequently encountered issues is an `ArrowIOError: Broken Pipe`. This error can happen in a couple of different ways. One of the most common ways this is encountered is from pressing **CTRL + C** sending a `KeyboardInterrupt` to Modin. In Ray, when a `KeyboardInterrupt` is sent, Ray will shutdown. This causes the `ArrowIOError: Broken Pipe` because there is no longer an available plasma store for working on remote tasks. This is working as intended, as it is not yet possible in Ray to kill a task that has already started computation.

The other common way this `Error` is encountered is to let your computer go to sleep. As an optimization, Ray will shutdown whenever the computer goes to sleep. This will result in the same issue as above, because there is no longer a running instance of the plasma store.

Solution

Restart your interpreter or notebook kernel.

Avoiding this Error

Avoid using `KeyboardInterrupt` and keeping your notebook or terminal running while your machine is asleep. If you do `KeyboardInterrupt`, you must restart the kernel or interpreter.

Error during execution: `ArrowInvalid: Maximum size exceeded (2GB)`

Encountering this issue means that the limits of the Arrow plasma store have been exceeded by the partitions of your data. This can be encountered during shuffling data or operations that require multiple datasets. This will only affect extremely large DataFrames, and can potentially be worked around by setting the number of partitions. This error is being actively worked on and should be resolved in a future release.

Solution

```
import modin.pandas as pd
pd.DEFAULT_NPARTITIONS = 2 * pd.DEFAULT_NPARTITIONS
```

This will set the number of partitions to a higher count, and reduce the size in each. If this does not work for you, please open an issue.

Hanging on `import modin.pandas as pd`

This can happen when Ray fails to start. It will keep retrying, but often it is faster to just restart the notebook or interpreter. Generally, this should not happen. Most commonly this is encountered when starting multiple notebooks or interpreters in quick succession.

Solution

Restart your interpreter or notebook kernel.

Avoiding this Error

Avoid starting many Modin notebooks or interpreters in quick succession. Wait 2-3 seconds before starting the next one.

Importing heterogeneous data by `read_csv`

Since Modin `read_csv` imports data in parallel, it can occur that data read by different partitions can have different type (this happens when columns contains heterogeneous data, i.e. column values are of different types), which are handled differntly. Example of such behaviour is shown below.

```
import os
import pandas
import modin.pandas as pd
from modin.config import NPartitions

NPartitions.put(2)

test_filename = "test.csv"
# data with heterogeneous values in the first column
data = """one,2
3,4
5,6
7,8
9.0,10
"""
kwargs = {
    # names of the columns to set, if `names` parameter is set,
    # header inffering from the first data row/rows will be disabled
    "names": ["col1", "col2"],

    # explicit setting of data type of column/columns with heterogeneous
    # data will force partitions to read data with correct dtype
    # "dtype": {"col1": str},
}


try :
    with open(test_filename, "w") as f:
        f.write(data)
```

(continues on next page)

(continued from previous page)

```
    pandas_df = pandas.read_csv(test_filename, **kwargs)
    pd_df = pd.read_csv(test_filename, **kwargs)

    print(pandas_df)
    print(pd_df)
finally:
    os.remove(test_filename)

Output:

pandas_df:
  col1  col2
0  one     2
1    3     4
2    5     6
3    7     8
4  9.0    10

pd_df:
  col1  col2
0  one     2
1    3     4
2    5     6
3  7.0     8
4  9.0    10
```

In this case *DataFrame* read by pandas in the column `col1` contain only `str` data because of the first string value ("one"), that forced pandas to handle full column data as strings. Modin the fisrt partition (the first three rows) read data similary to pandas, but the second partition (the last two rows) doesn't contain any strings in the first column and it's data is read as floats because of the last column value and as a result *7* value was read as *7.0*, that differs from pandas output.

The above example showed the mechanism of occurence of pandas and Modin `read_csv` outputs discrepancy during heterogeneous data import. Please note, that similar situations can occur during different data/parameters combinations.

Solution

In the case if heterogeneous data is detected, corresponding warning will be showed in the user's console. Currently, the discrepancies of such type doesn't properly handled by Modin, and to avoid this issue, it is needed to set `dtype` parameter of `read_csv` function manually to force correct data type definition during data import by partitions. Note, that to avoid excessive performance degradation, `dtype` value should be set fine-grained as it possible (specify `dtype` parameter only for columns with heterogeneous data).

Setting of `dtype` parameter works well for most of the cases, but, unfortunetely, it is ineffective if data file contain column which should be interpreted as index (`index_col` parameter is used) since `dtype` parameter is responsible only for data fields. For example, if in the above example, `kwargs` will be set in the next way:

```
kwargs = {
    "names": ["col1", "col2"],
    "dtype": {"col1": str},
    "index_col": "col1",
}
```

Resulting Modin DataFrame will contain incorrect value as in the case if `dtype` is not set:

```
col1
one      2
3        4
5        6
7.0      8
9.0     10
```

In this case data should be imported without setting of `index_col` parameter and only then index column should be set as index (by using `DataFrame.set_index` funcion for example) as it is shown in the example below:

```
pd_df = pd.read_csv(filename, dtype=data_dtype, index_col=None)
pd_df = pd_df.set_index(index_col_name)
pd_df.index.name = None
```

Error when using OmniSci engine along with `pyarrow.gandiva`: `LLVM ERROR: inconsistency in registered CommandLine options`

This can happen when you use OmniSci engine along with `pyarrow.gandiva`:

```
import modin.config as cfg
cfg.Engine.put("Native")  # 'omniscidbe'/'dbe' would be imported with dlopen flags
cfg.StorageFormat.put("Omnisci")
cfg.IsExperimental.put(True)
import modin.pandas as pd
import pyarrow.gandiva as gandiva  # Error
CommandLine Error: Option 'enable-vfe' registered more than once!
LLVM ERROR: inconsistency in registered CommandLine options
Aborted (core dumped)
```

Solution

Do not use OmniSci engine along with `pyarrow.gandiva`.

3.27 Contact

3.27.1 Mailing List

https://groups.google.com/forum/#!forum/modin-dev

General questions, potential contributors, and ideas should be directed to the developer mailing list. It is an open Google Group, so feel free to join anytime! If you are unsure about where to ask or post something, the mailing list is a good place to ask as well.

3.27.2 Issues

https://github.com/modin-project/modin/issues

Bug reports and feature requests should be directed to the issues page of the Modin GitHub repo.

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

E

F

G

P

Q

R

T

U

V

W