Modin

Release 0.13.3+0.gbac4031.dirty

Modin contributors

Mar 18, 2022

CONTENTS

Note:

Estimated Reading Time: 10 minutes
You can follow along this tutorial in a Jupyter notebook here.

CHAPTER

ONE

INSTALLATION

Note:

Estimated Reading Time: 15 minutes

If you already installed Modin on your machine, you can skip this section.

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`.

1.1 Installing with pip

1.1.1 Stable version

Modin can be installed with `pip` on Linux, Windows and MacOS. To install the most recent stable release run the following:

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

Modin can be used with *Ray*, *Dask*, or *OmniSci* engines. 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! See below for OmniSci engine installation.

1.1.2 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.

1.1.3 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[ray]" # If you want to use the Ray execution engine
```

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

1.1.4 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.

1.2 Installing with conda

1.2.1 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

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
```

`conda` may be slow installing `modin-omnisci` and hence `modin-all` packages so it's worth trying to set `channel_priority` to `strict` prior the installation process:

```
conda config --set channel_priority strict
```

1.2.2 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 to get started can be found in the Intel Distribution of Modin Getting Started guide.

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.

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 worse than native Windows.

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 .
```

CHAPTER

TWO

USING MODIN

In this section, we show how Modin can be used to accelerate your pandas workflows on a single machine up to multiple machines in a cluster setting.

2.1 Using Modin Locally

Note:

Estimated Reading Time: 5 minutes

In our quickstart example, we have already seen how you can achieve considerable speedup from Modin, even on a single machine. Users do 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 existing pandas code** while experiencing a considerable speedup from Modin, even on a single machine.

To use Modin on a single machine, only a modification of the import statement is needed. 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.

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

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

2.1.1 Optional Configurations

When using Modin locally on a single machine or laptop (without a cluster), Modin will automatically create and manage a local Dask or Ray cluster for the executing your code. So when you run an operation for the first time with Modin, you will see a message like this, indicating that a Modin has automatically initialized a local cluster for you:

```
df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
```

```
  UserWarning: Ray execution environment not yet initialized. Initializing...
  To remove this warning, run the following python code before doing dataframe␣
↪operations:

   import ray
   ray.init()
```

(continues on next page)

(continued from previous page)

```
If you prefer to use Dask over Ray as your execution backend, you can use the
following code to modify the default configuration:
```

```
import modin
modin.config.Engine.put("Dask")
```

```
df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
```

```
UserWarning: Dask execution environment not yet initialized. Initializing...
To remove this warning, run the following python code before doing dataframe operations:

   from distributed import Client

   client = Client()
```

Finally, if you already have an Ray or Dask engine initialized, Modin will automatically attach to whichever engine is available. If you are interested in using Modin with OmniSci engine, please refer to *these instructions*. For additional information on other settings you can configure, see *this page* for more details.

2.1.2 Advanced: Configuring the resources Modin uses

Modin automatically check the number of CPUs available on your machine and sets the number of partitions to be equal to the number of CPUs. You can verify this by running the following code:

```
import modin
print(modin.config.NPartitions.get()) #prints 16 on a laptop with 16 physical cores
```

Modin fully utilizes the resources on your machine. To read more about how this works, see *this page* for more details.

Since Modin will use all of the resources available on your machine by default, at times, it is possible that you may like 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`:

```
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.

```
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).

Note: Make sure to update the `MODIN_CPUS` configuration and initialize your preferred engine before you start working with the first operation using Modin! Otherwise, Modin will opt for the default setting.

2.2 Using Modin in a Cluster

Note:

Estimated Reading Time: 15 minutes
You can follow along in a Jupyter notebook in this two-part tutorial: [Part 1], [Part 2].

Often in practice we have a need to exceed the capabilities of a single machine. Modin works and performs well in both local mode and in a cluster environment. The key advantage of Modin is that your notebook does not change between local development and cluster execution. Users are not required to think about how many workers exist or how to distribute and partition their data; Modin handles all of this seamlessly and transparently.

2.2.1 Starting up a Ray Cluster

Modin is able to utilize Ray's built-in autoscaled cluster. To launch a Ray cluster using Amazon Web Service (AWS), you can use this file as the config file.

```
pip install boto3
aws configure
```

To start up the Ray cluster, run the following command in your terminal:

```
ray up modin-cluster.yaml
```

This configuration script starts 1 head node (m5.24xlarge) and 7 workers (m5.24xlarge), 768 total CPUs. For more information on how to launch a Ray cluster across different cloud providers or on-premise, you can also refer to the Ray documentation here.

2.2.2 Connecting to a Ray Cluster

To connect to the Ray cluster, run the following command in your terminal:

```
ray attach modin-cluster.yaml
```

The following code checks that the Ray cluster is properly configured and attached to Modin:

```
import ray
ray.init(address="auto")
from modin.config import NPartitions
assert NPartitions.get() == 768, "Not all Ray nodes are started up yet"
ray.shutdown()
```

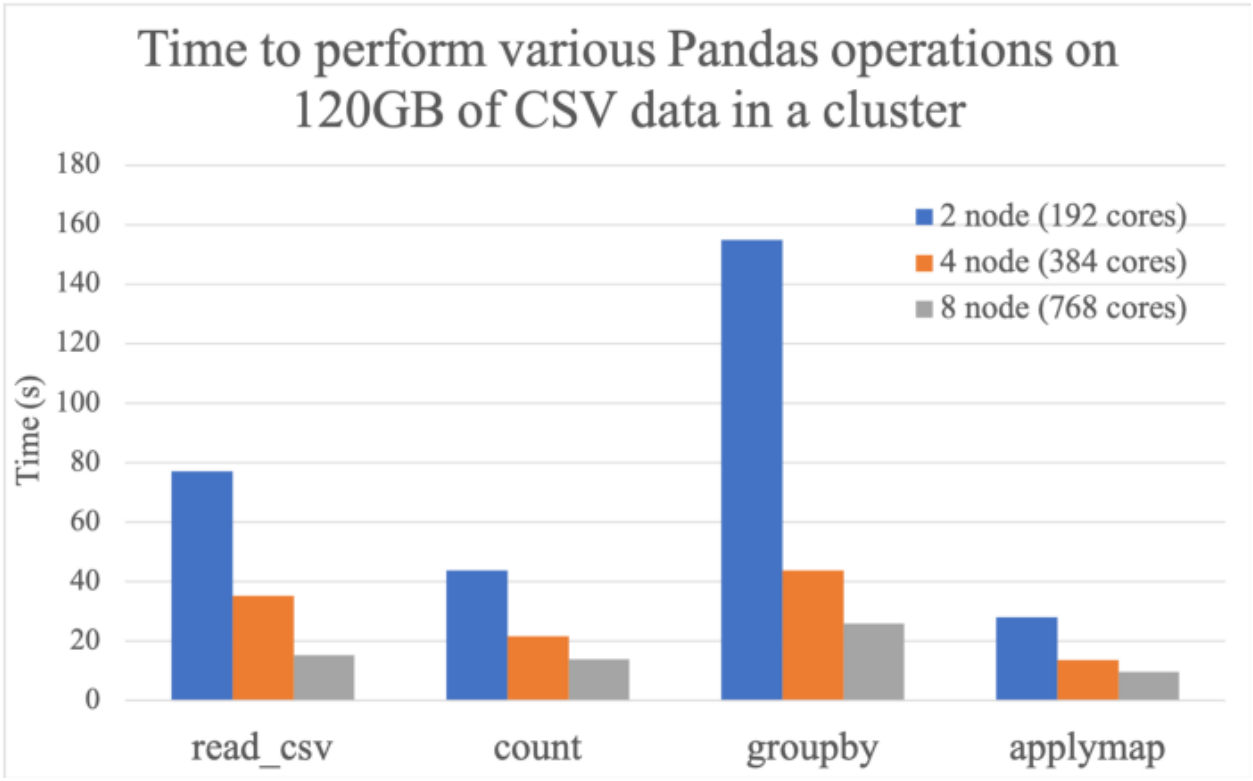
Congratualions! You have successfully connected to the Ray cluster. See more on the Modin in the Cloud documentation page.

2.2.3 Using Modin on a Ray Cluster

Now that we have a Ray cluster up and running, we can use Modin to perform pandas operation as if we were working with pandas on a single machine. We test Modin's performance on the 120GB NYC Taxi dataset that was provided as part of our cluster setup script. We can time the following operation in a Jupyter notebook:

```
%%time
df = pd.read_csv("big_yellow.csv", quoting=3)

%%time
count_result = df.count()

%%time
groupby_result = df.groupby("passenger_count").count()

%%time
apply_result = df.applymap(str)
```

Modin performance scales as the number of nodes and cores increases. The following chart shows the performance of the above operations with 2, 4, and 8 nodes, with improvements in performance as we increase the number of resources Modin can use.



2.2.4 Advanced: Configuring your Ray Environment

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!

CHAPTER

THREE

WHY MODIN?

In this section, we explain the design and motivation behind Modin and why you should use Modin to scale up your pandas workflows. We first describe the architectural differences between pandas and Modin. Then we describe how Modin can also help resolve out-of-memory issues common to pandas. Finally, we look at the key differences between Modin and other distributed dataframe libraries.

3.1 How does Modin differ from pandas?

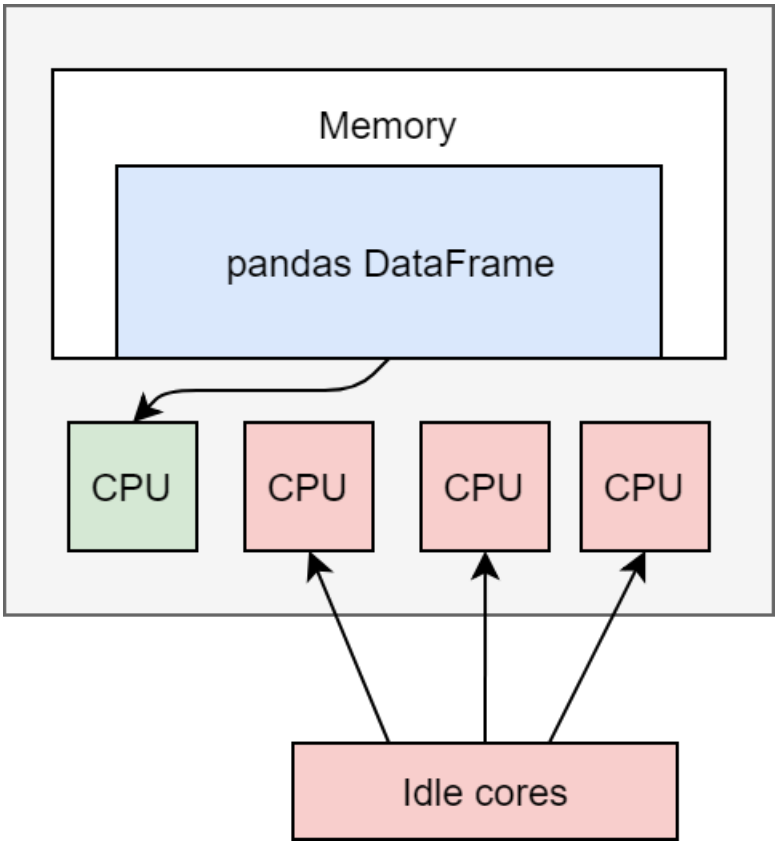
Note:

Estimated Reading Time: 10 minutes

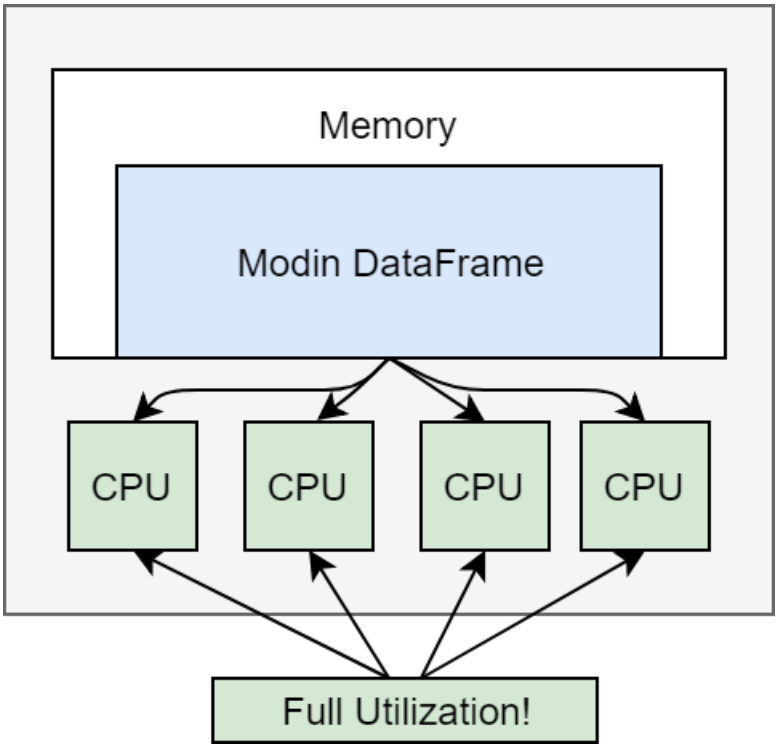
In the earlier tutorials, we have seen how Modin can be used to speed up pandas workflows. Here, we discuss at a high level how Modin works, in particular, how Modin's dataframe implementation differs from pandas.

3.1.1 Scalablity of implementation

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. 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:

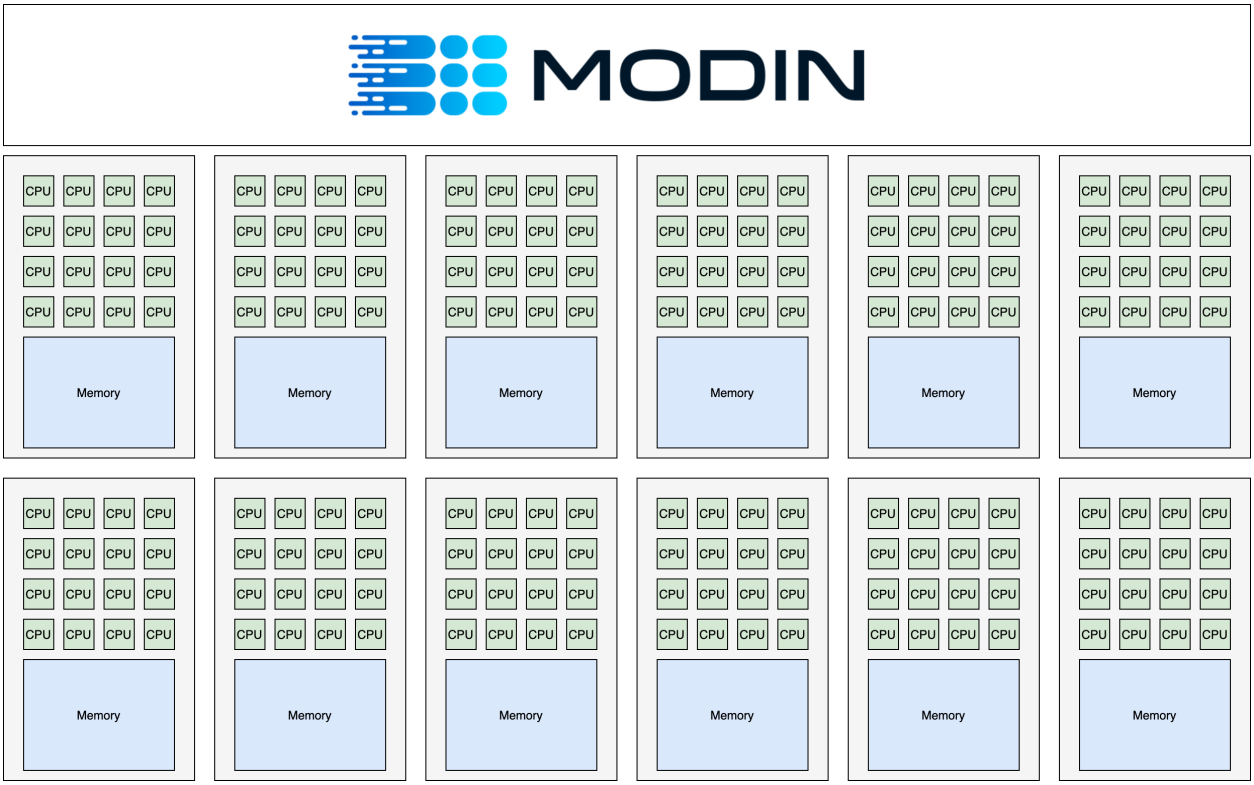


However, Modin's implementation enables you to use all of the cores on your machine, or all of the cores in an entire cluster. On a laptop, it will look something like this:



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!

3.1.2 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.

3.1.3 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 *architecture* documentation.

3.2 Out-of-memory data with Modin

Note:

Estimated Reading Time: 10 minutes

When using pandas, you might run into a memory error if you are working with large datasets that cannot fit in memory or perform certain memory-intensive operations (e.g., joins).

Modin solves this problem by spilling over to disk, in other words, it uses your disk as an overflow for memory so that you can work with datasets that are too large to fit in memory. By default, Modin leverages out-of-core methods to handle datasets that don't fit in memory for both Ray and Dask engines.

3.2.1 Motivating Example: Memory error with pandas

pandas makes use of in-memory data structures to store and operate on data, which means that if you have a dataset that is too large to fit in memory, it will cause an error on pandas. As an example, let's creates a 80GB DataFrame by appending together 40 different 2GB DataFrames.

```
import pandas
import numpy as np
df = pandas.concat([pandas.DataFrame(np.random.randint(0, 100, size=(2**20, 2**8))) for _
↪ in range(40)]) # Memory Error!
```

When we run this on a laptop with 32GB of RAM, pandas will run out of memory and throw an error (e.g., `MemoryError` , `Killed: 9`).

The pandas documentation has a great section on recommendations for scaling your analysis to these larger datasets. However, this generally involves loading in less data or rewriting your pandas code to process the data in smaller chunks.

3.2.2 Operating on out-of-memory data with Modin

In order to work with data that exceeds memory constraints, you can use Modin to handle these large datasets.

```
import modin.pandas as pd
import numpy as np
df = pd.concat([pd.DataFrame(np.random.randint(0, 100, size=(2**20, 2**8))) for _ in␣
↪range(40)]) # 40x2GB frames -- Working!
df.info()
```

Not only does Modin let you work with datasets that are too large to fit in memory, we can perform various operations on them without worrying about memory constraints.

3.2.3 Advanced: Configuring out-of-core settings

By default, out-of-core functionality 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 for more details.

3.3 Modin vs. Dask DataFrame vs. Koalas

Libraries such as Dask DataFrame (DaskDF for short) and Koalas aim to support the pandas API on top of distributed computing frameworks, Dask and Spark respectively. Instead, Modin aims to preserve the pandas API and behavior as is, while abstracting away the details of the distributed computing framework underneath. Thus, the aims of these libraries are fundamentally different.

Specifically, Modin enables pandas-like

- row and column-parallel operations, unlike DaskDF and Koalas that only support row-parallel operations
- indexing & ordering semantics, unlike DaskDF and Koalas that deviate from these semantics
- eager execution, unlike DaskDF and Koalas that provide lazy execution

As a result, Modin's coverage is more than 90% of the pandas API, while DaskDF and Koalas' coverage is about 55%.

For more technical details please see our VLDB 2022 research paper, referenced here.

3.3.1 Brief Overview of DaskDF and Koalas

Dask's DataFrame (DaskDF) is effectively a meta-DataFrame, partitioning and scheduling many smaller `pandas.DataFrame` objects. Users construct a task graph of dataframe computation step by step and then trigger computation using the `compute` function.

Spark's Koalas provides the pandas API on Spark, leveraging the preexisting Spark SQL optimizer to execute select pandas commands. Like DaskDF, Koalas also employs lazy computation, only triggering computation when the user requests to see the results.

3.3.2 Partitioning and Parallelization

Modin, DaskDF, Koalas are all examples of parallel dataframe systems. Parallelism is achieved by partitioning a large dataframe into smaller ones that can be operated on in parallel. As a result, the partitioning scheme chosen by the system dictates the pandas functions that can or can not be supported.

DaskDF and Koalas only support row-oriented partitioning and parallelism. This approach is analogous to relational databases. The dataframe is conceptually broken down into horizontal partitions along rows, where each partition is independently processed if possible. When DaskDF or Koalas are required to perform column-parallel operations

that to be done on columns independently (e.g., dropping columns with null values via `dropna` on the column `axis`), they either perform very poorly with no parallelism or do not support that operation.

Modin supports both row, column, and cell-oriented partitioning and parallelism. That is, the dataframe can be conceptually broken down as groups of rows, groups of columns, or both groups of rows and groups of columns (effectively a block or sub-matrix). Modin will transparently reshape the partitioning as necessary for the corresponding operation, based on whether the operation is row-parallel, column-parallel, or cell-parallel (independently applied to each unit cell). This allows Modin to support more of the pandas API and do so efficiently. Due to the finer-grained control over the partitioning, Modin can support a number of operations that are very challenging to parallelize in row-oriented systems (e.g., `transpose`, `median`, `quantile`). This flexibility in partitioning also gives Modin tremendous power to implement efficient straggler mitigation and improve utilization over the entire cluster.

3.3.3 API Coverage

One of the key benefits of pandas is its versatility, due to the wide array of operations, with more than 600+ API operations for data cleaning, feature engineering, data transformation, data summarization, data exploration, and machine learning. However, it is not trivial to develop scalable implementations of each of these operations in a dataframe system. **DaskDF and Koalas only implements about** 55% **of the pandas API**; they do not implement certain APIs that would deviate from the row-wise partitioning approach, or would be inefficient with the row-wise parallelization. For example, Dask does not implement `iloc`, `MultiIndex`, `apply(axis=0)`, `quantile` (only approximate quantile is available), `median`, and more. Given DaskDF's row-oriented architecture, `iloc`, for example, can technically be implemented, but it would be inefficient, and column-wise operations such as `apply(axis=0)` would be impossible to implement. Similarly, Koalas does not implement `apply(axis=0)` (it only applies the function per row partition, giving a different result), `quantile`, `median` (only approximate quantile/median is available), `MultiIndex`, `combine`, `compare` and more.

Modin supports all of the above pandas API functions, as well as others, with more than 90% **coverage of the pandas API.** Modin additionally acts as a drop-in replacement for pandas, such that even if the API is not yet supported, it still works by falling back to running vanilla 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 wishes to go back to running pandas directly, they are not locked in to using Modin and can switch between Modin and pandas at no cost. In other words, scripts and notebooks written in Modin can be converted to and from pandas as the user desires by simply replacing the import statement.

3.3.4 Execution Semantics

DaskDF and Koalas make use of lazy evaluation, which means that the computation is delayed until users explicitly evaluate the results. This mode of evaluation places a lot of optimization responsibility on the user, forcing them to think about when it would be useful to inspect the intermediate results or delay doing so. Specifically, DaskDF's API differs from pandas in that it requires users to explicitly call `.compute()` to materialize the result of the computation. Often if that computation corresponds to a long chain of operators, this call can take a very long time to execute. Overall, the need to explicitly trigger computation makes the API less convenient to work with, but gives DaskDF and Koalas the opportunity to perform holistic optimizations over the entire dataflow graph. However, to the best of our knowledge, neither DaskDF nor Koalas actually leverage holistic optimizations.

Modin employs eager evaluation, like pandas. Eager evaluation is the default mode of operation for data scientists when working with pandas in an interactive environment, such as Jupyter Notebooks. Modin reproduces this familiar behavior by performing all computations eagerly as soon as it is issued, so that users can inspect intermediate results and quickly see the results of their computations without having to wait or explicitly trigger computation. This is especially useful during interactive data analysis, where users often iterate on their dataframe workflows or build up their dataframe queries in an incremental fashion. Modin also supports lazy evaluation via the OmniSci engine, you can learn more about it here. We also have developed techniques for opportunistic evaluation that bridges the gap between lazy and eager evaluation that will be incorporated in Modin in the future.

3.3.5 Ordering Semantics

By default, pandas preserves the order of the dataframe, so that users can expect a consistent, ordered view as they are operating on their dataframe.

Both DaskDF and Koalas make no guarantees about the order of rows in the DataFrame. This is because DaskDF sorts the `index` for optimization purposes to speed up computations that involve the row index; and as a result, it does not support user-specified order. Likewise, Koalas does not support ordering by default because it will lead to a performance overhead when operating on distributed datasets.

DaskDF additionally does not support multi-indexing or sorting. DaskDF sorts the data based on a single set of row labels for fast row lookups, and builds an indexing structure based on these row labels. Data is both logically and physically stored in the same order. As a result, DaskDF does not support a *sort* function.

Modin reproduces the intuitive behavior in pandas where the order of the DataFrame is preserved, and supports multi-indexing. Enforcing ordering on a parallel dataframe system like Modin requires non-trivial effort that involves decoupling of the logical and physical representation of the data, enabling the order to be lazily kept up-to-date, but eagerly computed based on user needs (See Section 4.2 in our recent paper). Modin abstracts away the physical representation of the data and provides an ordered view that is consistent with user's expectations.

3.3.6 Compatibility with Computational Frameworks

DaskDF and Koalas are meant to be run on Dask and Spark respectively. They are highly tuned to the corresponding frameworks, and cannot be ported to other computational frameworks.

Modin's highly modular design is architected to run on a variety of systems, and support a variety of APIs. The goal for the extensible design is that users can take the same notebook or script and seamlessly move between different clusters and environments, with Modin being able to support the pandas API on your preexisting infrastructure. Currently, Modin support running on Dask's compute engine in addition to Ray. The modular design makes it easier for developers to different execution engines or compile to different memory formats. Modin can run on a Dask cluster in the same way that DaskDF can, but they differ in the ways described above. In addition, Modin is continually expanding to support popular data processing APIs (SQL in addition to pandas, among other DSLs for data processing) while leveraging the same underlying execution framework. Modin's flexible architecture also means that as the pandas API continues to evolve, Modin can quickly move towards supporting new versions of the pandas API.

3.3.7 Performance Comparison

On operations supported by all systems, Modin provides substantial speedups. Thanks to its optimized design, Modin is able to take advantage of multiple cores relative to both Koalas and DaskDF to efficiently execute pandas operations. It is notable that Koalas is often slower than pandas, due to the overhead of Spark.

Modin provides substantial speedups even on operators not supported by other systems. Thanks to its flexible partitioning schemes that enable it to support the vast majority of pandas operations — be it row, column, or cell-oriented - Modin provides benefits on operations such as `join`, `median`, and `infer_types`. While Koalas performs `join` slower than Pandas, Dask failed to support `join` on more than 20M rows, likely due poor support for shuffles. Details of the benchmark and additional join experiments can be found in our paper.

Modin is built on many years of research and development at UC Berkeley. For more information on how this works underneath the hoods, check out our publications in this space:

- Flexible Rule-Based Decomposition and Metadata Independence in Modin (VLDB 2021)
- Enhancing the Interactivity of Dataframe Queries by Leveraging Think Time (IEEE Data Eng 2021)

- Dataframe Systems: Theory, Architecture, and Implementation (PhD Dissertation 2021)
- Scaling Data Science does not mean Scaling Machines (CIDR 2021)
- Towards Scalable Dataframe Systems (VLDB 2020)

CHAPTER

FOUR

EXAMPLES AND RESOURCES

Here you can find additional resources to learn about Modin. To learn more about advanced usage for Modin, please refer to *this section*.

4.1 Usage Examples

The following notebooks demonstrate how Modin can be used for scalable data science:

- Quickstart Guide to Modin [Source]
- Using Modin with the NYC Taxi Dataset [Source]
- Using Experimental Modin in the cloud with the NYC Taxi Dataset on a AWS cluster [Source]
- Modin for Machine Learning with scikit-learn [Source]

4.2 Tutorials

The following tutorials cover the basic usage of Modin. Here is a one hour video tutorial that walks through these basic exercises.

- Exercise 1: Introduction to Modin [Source]
- Exercise 2: Speed Improvements with Modin [Source]
- Exercise 3: Defaulting to pandas with Modin [Source]

The following tutorials covers more advanced features in Modin:

- Exercise 4: Experimental Features in Modin (Spreadsheet, Progress Bar) [Source]
- Exercise 5: Setting up Modin in a Cluster Environment [Source]
- Exercise 6: Running Modin in a Cluster Environment [Source]

4.3 Talks & Podcasts

- Scaling Interactive Data Science with Modin and Ray (20 minute, Ray Summit 2021)
- Unleash The Power Of Dataframes At Any Scale With Modin (40 minute, Python Podcast 2021)
- [Russian] Distributed Data Processing and XGBoost Training and Prediction with Modin (30 minute, PyCon Russia 2021)
- [Russian] Efficient Data Science with Modin (30 minute, ISP RAS Open 2021)
- Modin: Scaling the Capabilities of the Data Scientist, not the Machine (1 hour, RISE Camp 2020)
- Modin: Pandas Scalability with Devin Petersohn (1 hour, Software Engineering Daily Podcast 2020)
- Introduction to the DataFrame and Modin (20 minute, RISECamp 2019)
- Scaling Interactive Pandas Workflows with Modin (40 minute, PyData NYC 2018)

4.4 Community contributions

Here are some blogposts and articles about Modin:

- Anaconda Blog: Scale your pandas workflow with Modin by Vasilij Litvinov
- The Modin view of Scaling Pandas by Devin Petersohn
- Data Science at Scale with Modin by Areg Melik-Adamyan
- Speed up Pandas using Modin by Eric D. Brown, D.Sc.
- Explore Python Libraries: Make Your DataFrames Parallel With Modin by Zachary Bennett
- Get faster pandas with Modin, even on your laptops by Parul Pandey
- How to speedup pandas by changing one line of code by Shrivarsheni
- How To Accelerate Pandas With Just One Line Of Code by Analytics India

Here are some articles contributed by the international community:

- [Chinese] Modin pandas by Python Chinese Community
- [German] Was ist Modin? by Dipl.-Ing. (FH) Stefan Luber
- [Russian] Pandas modin by
- [Korean] modin pandas by

If you would like your articles to be featured here, please submit a pull request to let us know!

CHAPTER

FIVE

FREQUENTLY ASKED QUESTIONS (FAQS)

Below, you will find answers to the most commonly asked questions about Modin. If you still cannot find the answer you are looking for, please post your question on the #support channel on our Slack community or open a Github issue.

5.1 FAQs: Why choose Modin?

5.1.1 What's wrong with pandas and why should I use Modin?

While pandas works extremely well on small datasets, as soon as you start working with medium to large datasets that are more than a few GBs, pandas can become painfully slow or run out of memory. This is because pandas is single-threaded. In other words, you can only process your data with one core at a time. This approach does not scale to larger data sets and adding more hardware does not lead to more performance gain.

The `DataFrame` is a highly scalable, parallel DataFrame. Modin transparently distributes the data and computation so that you can continue using the same pandas API while being able to work with more data faster. Modin lets you use all the CPU cores on your machine, and because it is lightweight, it often has less memory overhead than pandas. See this *page* to learn more about how Modin is different from pandas.

5.1.2 Why not just improve pandas?

pandas is a massive community and well established codebase. Many of the issues we have identified and resolved with pandas are fundamental to its current implementation. While we would be happy to donate parts of Modin that make sense in pandas, many of these components would require significant (or total) redesign of the pandas architecture. Modin's architecture goes beyond pandas, which is why the pandas API is just a thin layer at the user level. To learn more about Modin's architecture, see the *architecture* documentation.

5.1.3 How much faster can I go with Modin compared to pandas?

Modin is designed to scale with the amount of hardware available. Even in a traditionally serial task like `read_csv`, we see large gains by efficiently distributing the work across your entire machine. Because it is so light-weight, Modin provides speed-ups of up to 4x on a laptop with 4 physical cores. This speedup scales efficiently to larger machines with more cores. We have several published papers that include performance results and comparisons against pandas.

5.1.4 How much more data would I be able to process with Modin?

Often data scientists have to use different tools for operating on datasets of different sizes. This is not only because processing large dataframes is slow, but also pandas does not support working with dataframes that don't fit into the available memory. As a result, pandas workflows that work well for prototyping on a few MBs of data do not scale to tens or hundreds of GBs (depending on the size of your machine). Modin supports operating on data that does not fit in memory, so that you can comfortably work with hundreds of GBs without worrying about substantial slowdown or memory errors. For more information, see *out-of-memory support* for Modin.

5.1.5 How does Modin compare to Dask DataFrame and Koalas?

TLDR: Modin has better coverage of the pandas API, has a flexible backend, better ordering semantics, and supports both row and column-parallel operations. Check out this *page* detailing the differences!

5.1.6 How does Modin work under the hood?

Modin is logically separated into different layers that represent the hierarchy of a typical Database Management System. User queries which perform data transformation, data ingress or data egress pass through the Modin Query Compiler which translates queries from the top-level pandas API Layer that users interact with to the Modin Core Dataframe layer. The Modin Core DataFrame is our efficient DataFrame implementation that utilizes a partitioning schema which allows for distributing tasks and queries. From here, the Modin DataFrame works with engines like Ray or Dask to execute computation, and then return the results to the user.

For more details, take a look at our system *architecture*.

5.2 FAQs: How to use Modin?

5.2.1 If I'm only using my laptop, can I still get the benefits of Modin?

Absolutely! 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 and allows you to work on data that doesn't fit in your laptop's RAM.

5.2.2 How do I use Jupyter or Colab notebooks with Modin?

You can take a look at this Google Colab installation guide and this notebook tutorial. Once Modin is installed, simply replace your pandas import with Modin import:

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

5.2.3 Which execution engine (Ray or Dask) should I use for Modin?

Modin lets you effortlessly speed up your pandas workflows with either Ray's or Dask's execution engine. You don't need to know anything about either engine in order to use it with Modin. If you only have one engine installed, Modin will automatically detect which engine you have installed and use that for scheduling computation. If you don't have a preference, we recommend starting with Modin's default Ray engine. If you want to use a specific compute engine, you can set the environment variable `MODIN_ENGINE` and Modin will do computation with that engine:

```
pip install "modin[ray]" # Install Modin dependencies and Ray to run on Ray
export MODIN_ENGINE=ray  # Modin will use Ray

pip install "modin[dask]" # Install Modin dependencies and Dask to run on Dask
export MODIN_ENGINE=dask  # Modin will use Dask
```

This can also be done with:

```
from modin.config import Engine

Engine.put("ray")  # Modin will use Ray
Engine.put("dask")  # Modin will use Dask
```

We also have an experimental OmniSciDB-based engine of Modin you can read about *here*. We plan to support more execution engines in future. If you have a specific request, please post on the #feature-requests channel on our Slack community.

5.2.4 How do I connect Modin to a database via *read_sql*?

To 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.

5.2.5 How can I contribute to Modin?

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

If you are interested in contributing please check out the *Contributing Guide* and then refer to the *Development Documentation*, where you can find system architecture, internal implementation details, and other useful information. Also check out the Github to view open issues and make contributions.

CHAPTER

SIX

TROUBLESHOOTING

We hope your experience with Modin is bug-free, but there are some quirks about Modin that may require troubleshooting. If you are still having issues, please post on the #support channel on our Slack community or open a Github issue.

6.1 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.

6.1.1 Warning during execution: `defaulting to pandas`

Please note, that while Modin covers a large portion of the pandas API, not all functionality is implemented. For methods that are not yet implemented, such as `asfreq`, you may see the following:

```
UserWarning: `DataFrame.asfreq` defaulting to pandas implementation.
```

To understand which functions will lead to this warning, we have compiled a list of *currently supported methods*. When you see this warning, Modin defaults to pandas by converting the Modin dataframe to pandas to perform the operation. Once the operation is complete in pandas, it is converted back to a Modin dataframe. These operations will have a high overhead due to the communication involved and will take longer than pandas. When this is happening, a warning will be given to the user to inform them that this operation will take longer than usual. You can learn more about this *here*.

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.

6.1.2 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.

6.1.3 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)

    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
```

(continues on next page)

(continued from previous page)

```
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
```

6.1.4 Using Modin with python multiprocessing

We strongly recommend not to mix the use of Modin with Ray or Dask engine selected in conjunction with python multiprocessing because that can lead to undefined behavior. One of such examples is shown below:

```
import modin.pandas as pd

# Ray engine is used by default
df = pandas.DataFrame([1, 2, 3])

def f(arg):
  return df + arg

if __name__ == '__main__':
  from multiprocessing import Pool

  with Pool(5) as p:
      print(p.map(f, [1]))
```

Even if this example may work on your machine, we do not recommend similar scenarios. The python multiprocessing will cause conflicts with excessive resource use by launching duplicated Ray clusters on the same machine.

6.2 Common errors

6.2.1 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`.

6.2.2 Error when using Dask engine: `RuntimeError: if __name__ == '__main__':`

The following *script.py* uses Modin with Dask as an execution engine and produces errors:

```
# script.py
import modin.pandas as pd
import modin.config as cfg

cfg.Engine.put("dask")

df = pd.DataFrame([0,1,2,3])
print(df)
```

A part of the produced errors by the script above would be the following:

```
File "/path/python3.9/multiprocessing/spawn.py", line 134, in _check_not_importing_main
  raise RuntimeError('''
  RuntimeError:
      An attempt has been made to start a new process before the
      current process has finished its bootstrapping phase.

      This probably means that you are not using fork to start your
      child processes and you have forgotten to use the proper idiom
      in the main module:

          if __name__ == '__main__':
              freeze_support()
              ...

      The "freeze_support()" line can be omitted if the program
      is not going to be frozen to produce an executable.
```

This happens because Dask Client uses fork to start processes.

Solution

To avoid the problem Dask Client creation needs to be moved into `__main__` scope of the module.

The corrected *script.py* would look like:

```
# script.py
import modin.pandas as pd
import modin.config as cfg

cfg.Engine.put("dask")

if __name__ == "__main__":
  df = pd.DataFrame([0, 1, 2, 3]) # Dask Client creation is hidden in the first call of␣
↪Modin functionality.
  print(df)
```

or

```
# script.py
from distributed import Client
```

(continues on next page)

(continued from previous page)

```
import modin.pandas as pd
import modin.config as cfg

cfg.Engine.put("dask")

if __name__ == "__main__":
  client = Client() # Explicit Dask Client creation.
  df = pd.DataFrame([0, 1, 2, 3])
  print(df)
```

CHAPTER

SEVEN

QUICK START GUIDE

To install the most recent stable release for Modin run the following:

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

For further instructions on how to install Modin with conda or for specific platforms or engines, see our detailed installation guide.

Modin acts as a drop-in replacement for pandas so you simply have to replace the import of pandas with the import of Modin as follows to speed up your pandas workflows:

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

CHAPTER

EIGHT

EXAMPLE: INSTANT SCALABILITY WITH NO EXTRA EFFORT

When working on large datasets, pandas becomes painfully slow or *runs out of memory*. Modin automatically scales up your pandas workflows by parallelizing the dataframe operations, so that you can more effectively leverage the compute resources available.

For the purpose of demonstration, we will load in modin as `pd` and pandas as `pandas`.

```
import modin.pandas as pd
import pandas

##############################################
### For the purpose of timing comparisons ###
##############################################
import time
import ray
ray.init()
##############################################
```

In this toy example, we look at the NYC taxi dataset, which is around 120MB in size. You can download this dataset to run the example locally.

```
# This may take a few minutes to download
import urllib.request
s3_path = "https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2021-01.csv"
urllib.request.urlretrieve(s3_path, "taxi.csv")
```

8.1 Faster Data Loading with `read_csv`

```
start = time.time()

pandas_df = pandas.read_csv(s3_path, parse_dates=["tpep_pickup_datetime", "tpep_dropoff_
↪datetime"], quoting=3)

end = time.time()
pandas_duration = end - start
print("Time to read with pandas: {} seconds".format(round(pandas_duration, 3)))
```

By running the same command `read_csv` with Modin, we generally get around 4X speedup for loading in the data in parallel.

```
start = time.time()

modin_df = pd.read_csv(s3_path, parse_dates=["tpep_pickup_datetime", "tpep_dropoff_
↪datetime"], quoting=3)

end = time.time()
modin_duration = end - start
print("Time to read with Modin: {} seconds".format(round(modin_duration, 3)))

print("Modin is {}x faster than pandas at `read_csv`!".format(round(pandas_duration /␣
↪modin_duration, 2)))
```

8.2 Faster `concat` across multiple dataframes

Our previous `read_csv` example operated on a relatively small dataframe. In the following example, we duplicate the same taxi dataset 100 times and then concatenate them together, resulting in a dataset around 19GB in size.

```
start = time.time()

big_pandas_df = pandas.concat([pandas_df for _ in range(25)])

end = time.time()
pandas_duration = end - start
print("Time to concat with pandas: {} seconds".format(round(pandas_duration, 3)))
```

```
start = time.time()

big_modin_df = pd.concat([modin_df for _ in range(25)])

end = time.time()
modin_duration = end - start
print("Time to concat with Modin: {} seconds".format(round(modin_duration, 3)))

print("Modin is {}x faster than pandas at `concat`!".format(round(pandas_duration /␣
↪modin_duration, 2)))
```

Modin speeds up the `concat` operation by more than 60X, taking less than a second to create the large dataframe, while pandas took close to a minute.

8.3 Faster `apply` over a single column

The performance benefits of Modin becomes aparent when we operate on large gigabyte-scale datasets. For example, let's say that we want to round up the number across a single column via the `apply` operation.

```
start = time.time()
rounded_trip_distance_pandas = big_pandas_df["trip_distance"].apply(round)

end = time.time()
```

(continues on next page)

(continued from previous page)

```
pandas_duration = end - start
print("Time to apply with pandas: {} seconds".format(round(pandas_duration, 3)))
```

```
start = time.time()

rounded_trip_distance_modin = big_modin_df["trip_distance"].apply(round)

end = time.time()
modin_duration = end - start
print("Time to apply with Modin: {} seconds".format(round(modin_duration, 3)))

print("Modin is {}x faster than pandas at `apply` on one column!".format(round(pandas_
↪duration / modin_duration, 2)))
```

Modin is more than 30X faster at applying a single column of data, operating on 130+ million rows in a second.

CHAPTER

NINE

SUMMARY

Hopefully, this tutorial demonstrated how Modin delivers significant speedup on pandas operations without the need for any extra effort. Throughout example, we moved from working with 100MBs of data to 20GBs of data all without having to change anything or manually optimize our code to achieve the level of scalable performance that Modin provides.

Note that in this quickstart example, we've only shown `read_csv`, `concat`, `apply`, but these are not the only pandas operations that Modin optimizes for. In fact, Modin covers more than 90% of the pandas API, yielding considerable speedups for many common operations.

CHAPTER

TEN

USAGE GUIDE

This guide describes both basic and advanced Modin usage, including usage examples, details regarding Modin configuration settings, as well as tips and tricks on how you can further optimize the performance of your workload with Modin.

10.1 Modin Configuration Settings

To adjust Modin's default behavior, you can set the value of Modin configs by setting an environment variable or by using the `modin.config` API. To list all avaliable configs in Modin, please run `python -m modin.config` to print all Modin configs with descriptions.

10.1.1 Public API

Potentially, the source of configs can be any, but for now only environment variables are implemented. Any environment variable originate from *EnvironmentVariable*, which contains most of the config API implementation.

class modin.config.envvars.**EnvironmentVariable**
Base class for environment variables-based configuration.

classmethod get()
Get config value.

Returns Decoded and verified config value.

Return type Any

classmethod get_help() → str
Generate user-presentable help for the config.

Return type str

classmethod get_value_source()
Get value source of the config.

Return type int

classmethod once(*onvalue*, *callback*)
Execute *callback* if config value matches *onvalue* value.

Otherwise accumulate callbacks associated with the given *onvalue* in the *_once* container.

Parameters

- **onvalue** (*Any*) – Config value to set.
- **callback** (*callable*) – Callable that should be executed if config value matches *onvalue*.

classmethod put(*value*)
Set config value.

Parameters **value** (*Any*) – Config value to set.

classmethod subscribe(*callback*)
Add *callback* to the *_subs* list and then execute it.

Parameters **callback** (*callable*) – Callable to execute.

10.1.2 Modin Configs List

Config Name	Env. Variable Name	Default Value	Description	Options
AsvDataSizeConfig	MODIN_ASV_DATASIZE_CONFIG		Allows to override default size of data (shapes).	
AsvImplementation	MODIN_ASV_USE_IMPL	modin	Allows to select a library that we will use for testing performance.	('modin', 'pandas')
BenchmarkMode	MODIN_BENCHMARK_MODE	False	Whether or not to perform computations synchronously.	
CpuCount	MODIN_CPUS	2	How many CPU cores to use during initialization of the Modin engine.	
DoLogRpyc	MODIN_LOG_RPYC		Whether to gather RPyC logs (applicable for remote context).	
DoTraceRpyc	MODIN_TRACE_RPYC		Whether to trace RPyC calls (applicable for remote context).	
DoUseCalcite	MODIN_USE_CALCITE	True	Whether to use Calcite for OmniSci queries execution.	
Engine	MODIN_ENGINE	Ray	Distribution engine to run queries by.	('Ray', 'Dask', 'Python', 'Native')
GpuCount	MODIN_GPUS		How may GPU devices to utilize across the whole distribution.	
IsDebug	MODIN_DEBUG		Force Modin engine to be "Python" unless specified by $MODIN_ENGINE.	
IsExperimental	MODIN_EXPERIMENTAL		Whether to Turn on experimental features.	
IsRayCluster	MODIN_RAY_CLUSTER		Whether Modin is running on pre-initialized Ray cluster.	
Memory	MODIN_MEMORY		How much memory (in bytes) give to an execution engine. • In Ray case: the	

10.1.3 Usage Guide

See example of interaction with Modin configs below, as it can be seen config value can be set either by setting the environment variable or by using config API.

```
import os

# Setting `MODIN_STORAGE_FORMAT` environment variable.
# Also can be set outside the script.
os.environ["MODIN_STORAGE_FORMAT"] = "OmniSci"

import modin.config
import modin.pandas as pd

# Checking initially set `StorageFormat` config,
# which corresponds to `MODIN_STORAGE_FORMAT` environment
# variable
print(modin.config.StorageFormat.get()) # prints 'Omnisci'

# Checking default value of `NPartitions`
print(modin.config.NPartitions.get()) # prints '8'

# Changing value of `NPartitions`
modin.config.NPartitions.put(16)
print(modin.config.NPartitions.get()) # prints '16'
```

10.2 Modin Usage Examples

This section shows Modin usage examples in different scenarios like Modin on a local/remote cluster, Modin in the cloud, the use of Modin spreadsheet.

10.2.1 Tutorials

The following tutorials cover the basic usage of Modin. Here is a one hour video tutorial that walks through these basic exercises.

- Exercise 1: Introduction to Modin [Source]
- Exercise 2: Speed Improvements with Modin [Source]
- Exercise 3: Defaulting to pandas with Modin [Source]

The following tutorials covers more advanced features in Modin:

- Exercise 4: Experimental Features in Modin (Spreadsheet, Progress Bar) [Source]
- Exercise 5: Setting up Modin in a Cluster Environment [Source]
- Exercise 6: Running Modin in a Cluster Environment [Source]

10.2.2 Data Science Benchmarks

- Using Modin with the NYC Taxi Dataset [Source]
- Using Modin with the Census Dataset (coming soon...)
- Using Modin with the Plasticc Dataset (coming soon...)

10.2.3 Modin in the Cloud

- Using Experimental Modin in the cloud with the NYC Taxi Dataset on an AWS cluster [Source]

10.2.4 Modin Spreadsheets

- Using Modin along with the Spreadsheets API [Source]

10.2.5 Modin with scikit-learn

- Modin for Machine Learning with scikit-learn [Source]

10.3 Advanced Usage

10.3.1 Pandas partitioning 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)
```

10.3.2 Modin Spreadsheets API

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
```

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.

```
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()
```

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.

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.

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

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 col-

umn 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,
    '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.

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.

10.3.3 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.

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()
```

10.3.4 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.

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)
```

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
```

(continues on next page)

(continued from previous page)

```
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
    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.

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.

10.3.5 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.

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
```

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

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.

Return type Column labels.

property feature_types
Get column types.

Return type Column types.

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

Return type dict

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

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

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

num_col()
Get number of columns.

Return type int

num_row()
Get number of rows.

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.

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.

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()

# Run training
```

(continues on next page)

(continued from previous page)

```
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}')
```

10.3.6 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.

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)

Setup environment

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

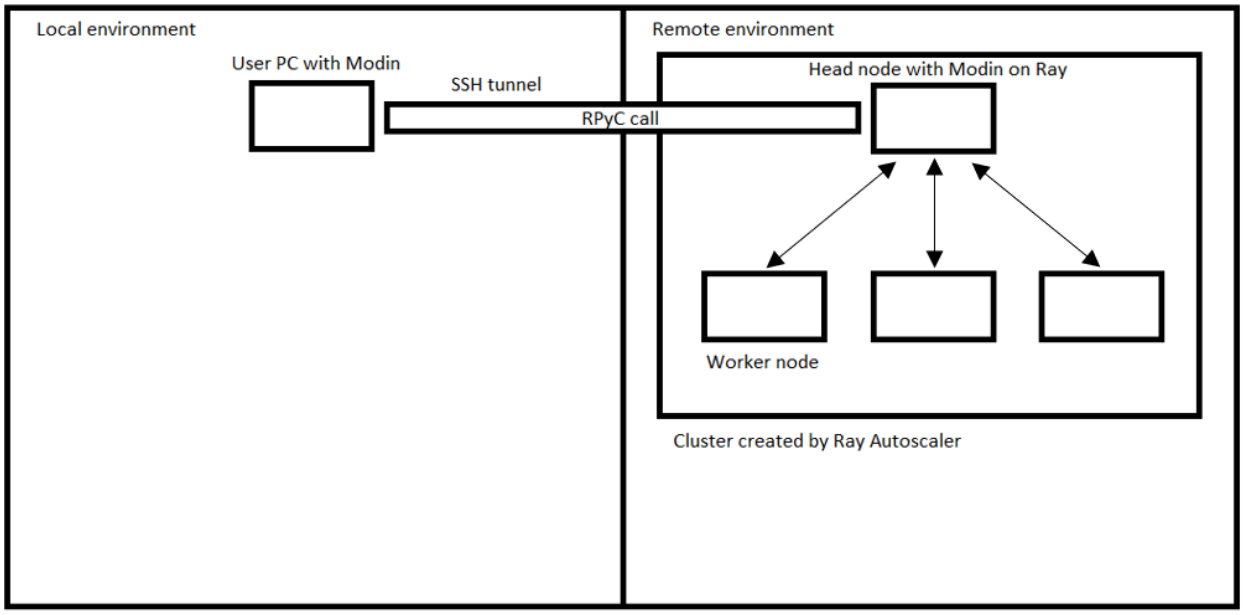
This command install the following dependencies:

- 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.

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.

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.

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.

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 us on Slack and Discourse for the latest updates!

10.3.7 Experimental APIs

Modin also supports these experimental APIs on top of pandas that are under active development.

- `read_csv_glob()` – read multiple files in a directory
- `read_sql()` – add optional parameters for the database connection
- `read_pickle_distributed()` – read multiple files in a directory
- `to_pickle_distributed()` – write to multiple files in a directory

10.3.8 DataFrame partitioning API

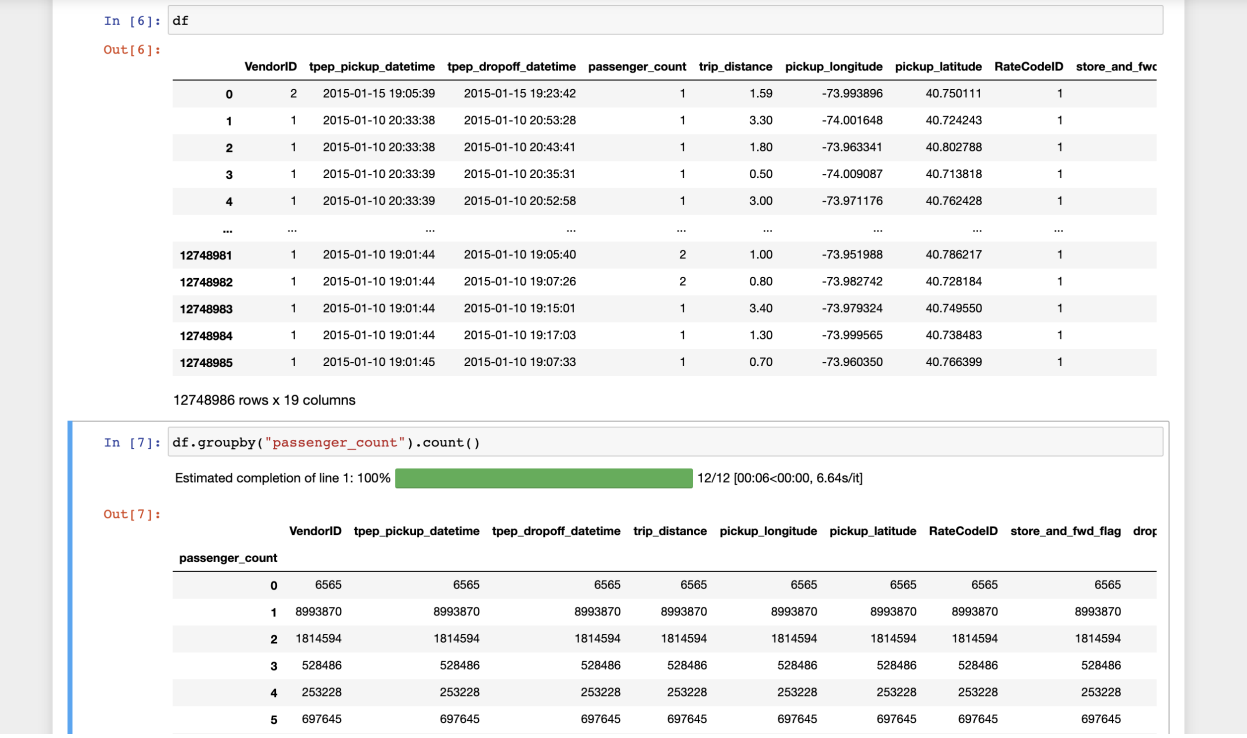
Modin DataFrame provides an API to directly access partitions: you can extract physical partitions from a *DataFrame*, modify their structure by reshuffling or applying some functions, and create a DataFrame from those modified partitions. Visit *pandas partitioning API* documentation to learn more.

10.3.9 Modin Spreadsheet API

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.

10.3.10 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	

10.3.11 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!

10.3.12 SQL on Modin Dataframes

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

```
# 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)
```

10.3.13 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.

10.4 Optimization Notes

Modin has chosen default values for a lot of the configurations here that provide excellent performance in most cases. This page is for those who love to optimize their code and those who are curious about existing optimizations within Modin. Here you can find more information about Modin's optimizations both for a pipeline of operations as well as for specific operations.

10.4.1 Understanding Modin's partitioning mechanism

Modin's partitioning is crucial for performance; so we recommend expert users to understand Modin's partitioning mechanism and how to tune it in order to achieve better performance.

How Modin partitions a dataframe

Modin uses a partitioning scheme that partitions a dataframe along both axes, resulting in a matrix of partitions. The row and column chunk sizes are computed independently based on the length of the appropriate axis and Modin's special *configuration variables* (`NPartitions` and `MinPartitionSize`):

- `NPartitions` is the maximum number of splits along an axis; by default, it equals to the number of cores on your local machine or cluster of nodes.
- `MinPartitionSize` is the minimum number of rows/columns to do a split. For instance, if `MinPartitionSize` is 32, the column axis will not be split unless the amount of columns is greater than 32. If it is is greater, for example, 34, then the column axis is sliced into two partitions: containing 32 and 2 columns accordingly.

Beware that `NPartitions` specifies a limit for the number of partitions *along a single axis*, which means, that the actual limit for the entire dataframe itself is the square of `NPartitions`.

Full-axis functions

Some of the aggregation functions require knowledge about the entire axis, for example at `.apply(foo, axis=0)` the passed function `foo` expects to receive data for the whole column at once.

When a full-axis function is applied, the partitions along this axis are collected at a single worker that processes the function. After the function is done, the partitioning of the data is back to normal.

Note that the amount of remote calls is equal to the number of partitions, which means that since the number of partitions is decreased for full-axis functions it also decreases the potential for parallelism.

Also note, that reduce functions such as `.sum()`, `.mean()`, `.max()`, etc, are not considered to be full-axis, so they do not suffer from the decreasing level of parallelism.

How to tune partitioning

As you can see from the examples above, the more the dataframe's shape is closer to a square, the closer the number of partitions to the square of `NPartitions`. In the case of `NPartitions` equals to the number of workers, that means that a single worker is going to process multiple partitions at once, which slows down overall performance.

If your workflow mainly operates with wide dataframes and non-full-axis functions, it makes sense to reduce the `NPartitions` value so a single worker would process a single partition.

Copy-pastable example, showing how tuning `NPartitions` value for wide frames may improve performance on your machine:

```
from multiprocessing import cpu_count
from modin.distributed.dataframe.pandas import unwrap_partitions
import modin.config as cfg
import modin.pandas as pd
import numpy as np
import timeit

# Generating data for a square-like dataframe
data = np.random.randint(0, 100, size=(5000, 5000))

# Explicitly setting `NPartitions` to its default value
cfg.NPartitions.put(cpu_count())

# Each worker processes `cpu_count()` amount of partitions
df = pd.DataFrame(data)
print(f"NPartitions: {cfg.NPartitions.get()}")
# Getting raw partitions to count them
partitions_shape = np.array(unwrap_partitions(df)).shape
print(
    f"The frame has {partitions_shape[0]}x{partitions_shape[1]}={np.prod(partitions_
↪shape)} partitions "
    f"when the CPU has only {cpu_count()} cores."
)
print(f"10 times of .abs(): {timeit.timeit(lambda: df.abs(), number=10)}s.")
# Possible output:
#   NPartitions: 112
#   The frame has 112x112=12544 partitions when the CPU has only 112 cores.
#   10 times of .abs(): 23.64s.

# Taking a square root of the the current `cpu_count` to make more even partitioning
cfg.NPartitions.put(int(cpu_count() ** 0.5))

# Each worker processes a single partition
df = pd.DataFrame(data)
print(f"NPartitions: {cfg.NPartitions.get()}")
# Getting raw partitions to count them
partitions_shape = np.array(unwrap_partitions(df)).shape
print(
    f"The frame has {partitions_shape[0]}x{partitions_shape[1]}={np.prod(partitions_
↪shape)} "
    f"when the CPU has {cpu_count()} cores."
```

(continues on next page)

(continued from previous page)

```
)
print(f"10 times of .abs(): {timeit.timeit(lambda: df.abs(), number=10)}s.")
# Possible output:
#   NPartitions: 10
#   The frame has 10x10=100 partitions when the CPU has 112 cores.
#   10 times of .abs(): 0.25s.
```

10.4.2 Avoid iterating over Modin DataFrame

Use `df.apply()` or other aggregation methods when possible instead of iterating over a dataframe. For-loops don't scale and forces the distributed data to be collected back at the driver.

Copy-pastable example, showing how replacing a for-loop to the equivalent `.apply()` may improve performance:

```
import modin.pandas as pd
import numpy as np
from timeit import default_timer as timer

data = np.random.randint(1, 100, (2 ** 10, 2 ** 2))

md_df = pd.DataFrame(data)

result = []
t1 = timer()
# Iterating over a dataframe forces to collect distributed data to the driver and doesn't␣
↪scale
for idx, row in md_df.iterrows():
    result.append((row[1] + row[2]) / row[3])
print(f"Filling a list by iterating a Modin frame: {timer() - t1:.2f}s.")
# Possible output: 36.15s.

t1 = timer()
# Using `.apply()` perfectly scales to all axis-partitions
result = md_df.apply(lambda row: (row[1] + row[2]) / row[3], axis=1).to_numpy().tolist()
print(f"Filling a list by using '.apply()' and converting the result to a list: {timer()␣
↪- t1:.2f}s.")
# Possible output: 0.22s.
```

10.4.3 Use Modin's Dataframe Algebra API to implement custom parallel functions

Modin provides a set of low-level parallel-implemented operators which can be used to build most of the aggregation functions. These operators are present in the algebra module. Modin DataFrame allows users to use their own aggregations built with this module. Visit the appropriate section of the documentation for the steps to do it.

10.4.4 Avoid mixing pandas and Modin DataFrames

Although Modin is considered to be a drop-in replacement for pandas, Modin and pandas are not intended to be used together in a single flow. Passing a pandas DataFrame as an argument for a Modin's DataFrame method may either slowdown the function (because it has to process non-distributed object) or raise an error. You would also get an undefined behavior if you pass a Modin DataFrame as an input to pandas methods, since pandas identifies Modin's objects as a simple iterable, and so can't leverage its benefits as a distributed dataframe.

Copy-pastable example, showing how mixing pandas and Modin DataFrames in a single flow may bottleneck performance:

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

data = np.random.randint(0, 100, (2 ** 20, 2 ** 2))

md_df, md_df_copy = pd.DataFrame(data), pd.DataFrame(data)
pd_df, pd_df_copy = pandas.DataFrame(data), pandas.DataFrame(data)

print("concat modin frame + pandas frame:")
# Concatenating modin frame + pandas frame using modin '.concat()'
# This case is bad because Modin have to process non-distributed pandas object
time = timeit.timeit(lambda: pd.concat([md_df, pd_df]), number=10)
print(f"\t{time}s.\n")
# Possible output: 0.44s.

print("concat modin frame + modin frame:")
# Concatenating modin frame + modin frame using modin '.concat()'
# This is an ideal case, Modin is being used as intended
time = timeit.timeit(lambda: pd.concat([md_df, md_df_copy]), number=10)
print(f"\t{time}s.\n")
# Possible output: 0.05s.

print("concat pandas frame + pandas frame:")
# Concatenating pandas frame + pandas frame using pandas '.concat()'
time = timeit.timeit(lambda: pandas.concat([pd_df, pd_df_copy]), number=10)
print(f"\t{time}s.\n")
# Possible output: 0.31s.

print("concat pandas frame + modin frame:")
# Concatenating pandas frame + modin frame using pandas '.concat()'
time = timeit.timeit(lambda: pandas.concat([pd_df, md_df]), number=10)
print(f"\t{time}s.\n")
# Possible output: TypeError
```

10.4.5 Operation-specific optimizations

merge

`merge` operation in Modin uses the broadcast join algorithm: combining a right Modin DataFrame into a pandas DataFrame and broadcasting it to the row partitions of the left Modin DataFrame. In order to minimize interprocess communication cost when doing an inner join you may want to swap left and right DataFrames.

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

left_data = np.random.randint(0, 100, size=(2**8, 2**8))
right_data = np.random.randint(0, 100, size=(2**12, 2**12))

left_df = pd.DataFrame(left_data)
right_df = pd.DataFrame(right_data)
%timeit left_df.merge(right_df, how="inner", on=10)
3.59 s  107 ms per loop (mean  std. dev. of 7 runs, 1 loop each)

%timeit right_df.merge(left_df, how="inner", on=10)
1.22 s  40.1 ms per loop (mean  std. dev. of 7 runs, 1 loop each)
```

Note that result columns order may differ for first and second `merge`.

CHAPTER

ELEVEN

SUPPORTED APIS

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.

To view the docs for the most recent release, check that you're viewing the stable version of the docs.

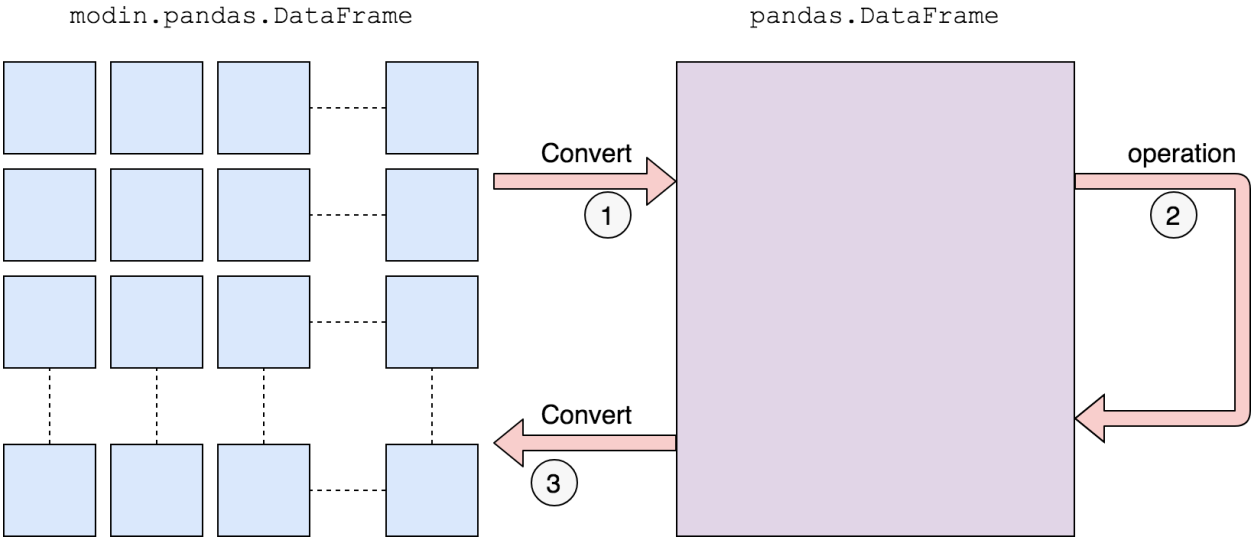
In order to install the latest version of Modin, follow the directions found on the installation page.

11.1 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.

11.1.1 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.

11.1.2 `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	
`asfreq`	asfreq	D	
`asof`	asof	Y	
`assign`	assign	Y	
`astype`	astype	Y	
`at`	at	Y	
`at_time`	at_time	Y	
`axes`	axes	Y	

continues on next page

Table 1 – continued from previous page

between_time	between_time	Y	
bfill	bfill	Y	
bool	bool	Y	
boxplot	boxplot	D	
clip	clip	Y	
combine	combine	Y	
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

continues on next page

Table 1 – continued from previous page

from_dict	from_dict	D	
from_records	from_records	D	
ge	ge	Y	See add
get	get	Y	
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_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_parquet	to_parquet	D	
to_period	to_period	D	
to_pickle	to_pickle	D	Experimental implementation: to_pickle_distributed
to_records	to_records	D	
to_sql	to_sql	Y	
to_stata	to_stata	D	

continues on next page

Table 1 – continued from previous page

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	

11.1.3 `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	
asfreq	D	
asobject	D	
asof	Y	
astype	Y	
at	Y	

Table 2 – continued from previous page

at_time	Y	
autocorr	Y	
axes	Y	
base	D	
between	D	
between_time	Y	
bfill	Y	
bool	Y	
cat	D	
clip	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	
expanding	D	
explode	Y	
factorize	D	
ffill	Y	
fillna	Y	
filter	Y	
first	Y	
first_valid_index	Y	
flags	D	

Table 2 – continued from previous page

floordiv	Y	
from_array	D	
ftype	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	
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	

Table 2 – continued from previous page

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	
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	

Table 2 – continued from previous page

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_dict	D	
to_excel	D	
to_frame	Y	
to_hdf	D	
to_json	D	
to_latex	D	
to_list	D	
to_numpy	D	
to_period	D	
to_pickle	D	
to_sql	Y	
to_string	D	
to_timestamp	D	
to_xarray	D	
tolist	D	
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	

Table 2 – continued from previous page

`var`	P	Modin defaults to pandas if given the `level` param.
`view`	D	
`where`	Y	

11.1.4 pandas Utilities Supported

If you run `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	

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

11.1.5 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	

CHAPTER

TWELVE

DEVELOPMENT

12.1 Contributing

12.1.1 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.

12.1.2 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`.

12.1.3 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.

12.1.4 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.

12.1.5 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
```

12.1.6 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.

12.1.7 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.

12.1.8 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.

12.1.9 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.

12.1.10 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*).

12.1.11 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.

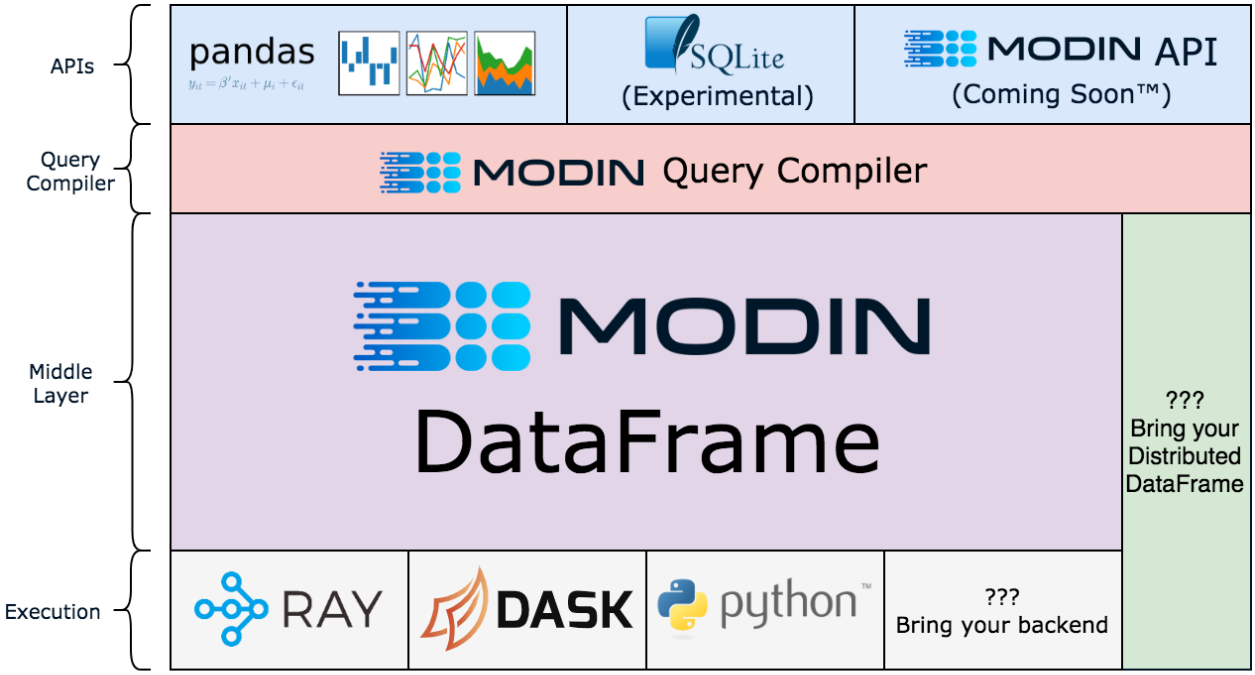
More docs on this coming soon...

12.2 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.

12.2.1 High-Level Architectural View

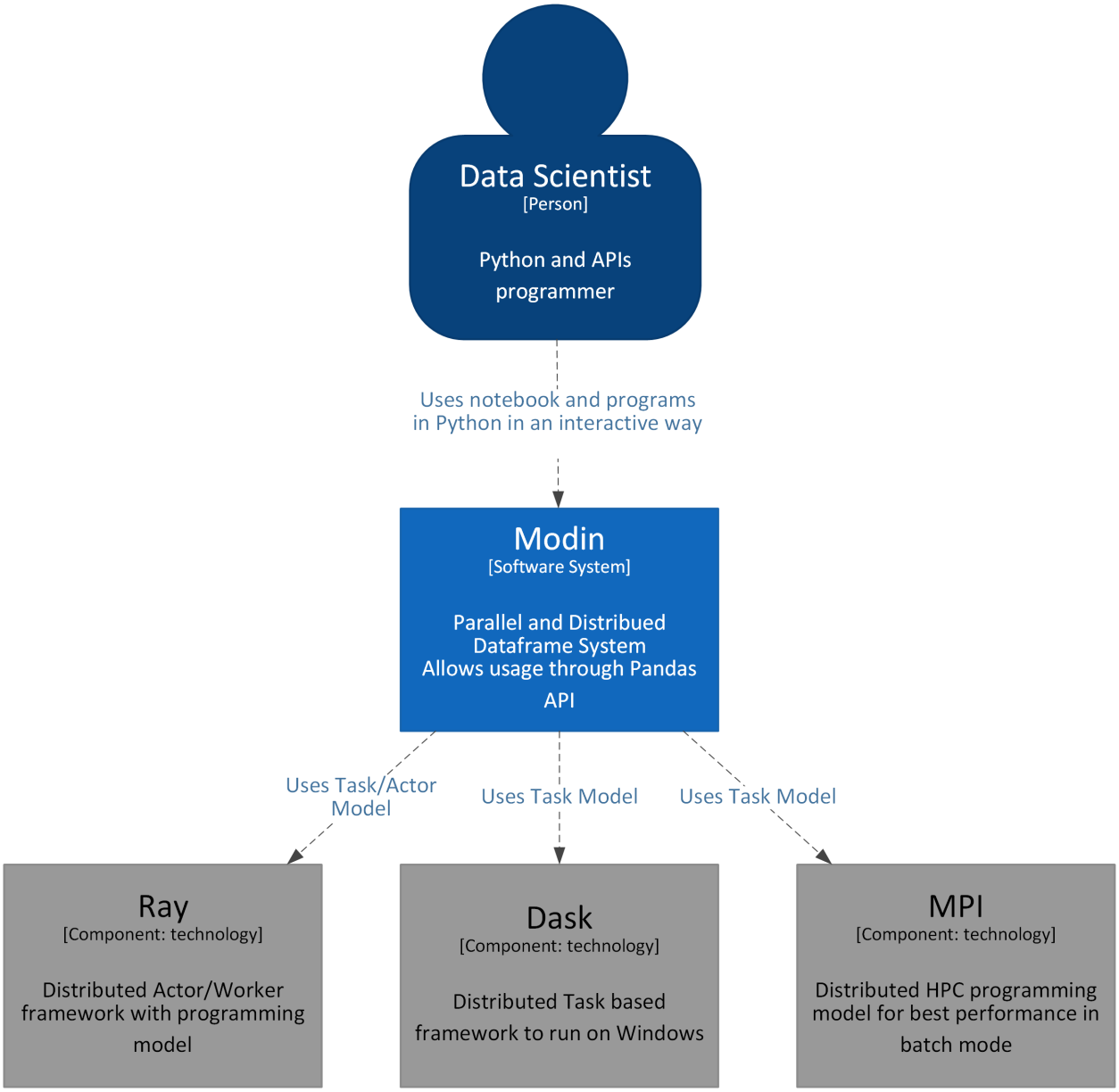
The diagram below outlines the general layered view to the components of Modin with a short description of each major section of the documentation following.



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.

12.2.2 System View

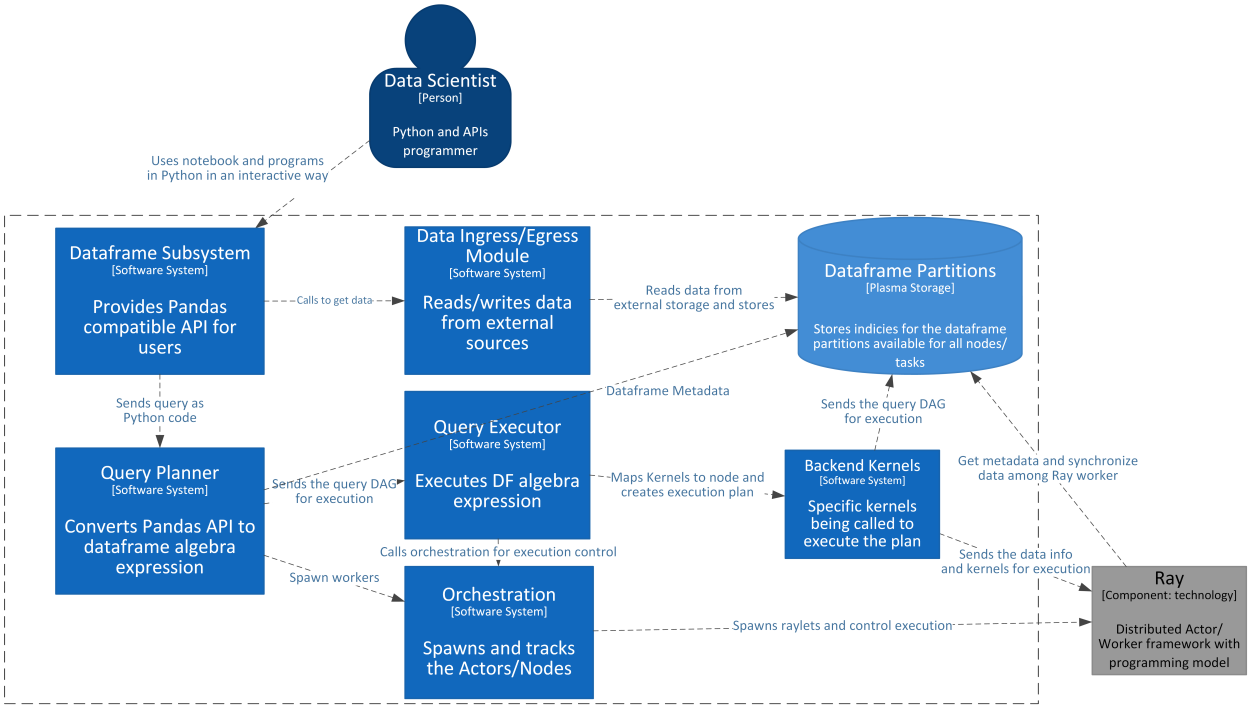
A top-down view of Modin's architecture is detailed below:



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.

12.2.3 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

12.2.4 Component View

User queries which perform data transformation, data ingress or data egress pass through the Modin components detailed below. The path the query takes is mostly similar across execution systems, with some minor exceptions like OmnisciOnNative.

Data Transformation

Query Compiler

The Query Compiler receives queries from the pandas API layer. The API layer is responsible for ensuring a clean input to the Query Compiler. The Query Compiler must have knowledge of the compute kernels and in-memory format of the data in order to efficiently compile the query.

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.

Core Modin Dataframe API

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 or Slack 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.

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.

Partitions

Partitions are responsible for managing a subset of the Dataframe. As mentioned below, 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, allowing Modin to exploit potential optimizations across both. These optimizations are explained further on the pages specific to the execution framework.

Execution Engine

This layer performs computation on partitions of the data. The Modin Dataframe is designed to work with task parallel frameworks, but integration with data parallel frameworks should be possible with some effort.

Storage Format

The storage format describes the in-memory partition type. The base storage format in Modin is pandas. In the default case, the Modin Dataframe operates on partitions that contain `pandas.DataFrame` objects.

Data Ingress

Note: Data ingress operations (e.g. `read_csv`) in Modin load data from the source into partitions and vice versa for data egress (e.g. `to_csv`) operation. Improved performance is achieved by reading/writing in partitions in parallel.

Data ingress starts with a function in the pandas API layer (e.g. `read_csv`). Then the user's query is passed to the Factory Dispatcher, which defines a factory specific for the execution. The factory for execution contains an IO class (e.g. `PandasOnRayIO`) whose responsibility is to perform a parallel read/write from/to a file. This IO class contains class methods with interfaces and names that are similar to pandas IO functions (e.g. `PandasOnRayIO.read_csv`). The IO class declares the Modin Dataframe and Query Compiler classes specific for the execution engine and storage format to ensure the correct object is constructed. It also declares IO methods that are mix-ins containing a combination of the engine-specific class for deploying remote tasks, the class for parsing the given file format and the class handling the chunking of the format-specific file on the head node (see dispatcher classes implementation details). The output from the IO class data ingress function is a Modin Dataframe.

Data Egress

Data egress operations (e.g. `to_csv`) are similar to data ingress operations up to execution-specific IO class functions construction. Data egress functions of the IO class are defined slightly different from data ingress functions and created only specifically for the engine since partitions already have information about its storage format. Using the IO class, data is exported from partitions to the target file.

Supported Execution Engines and Storage Formats

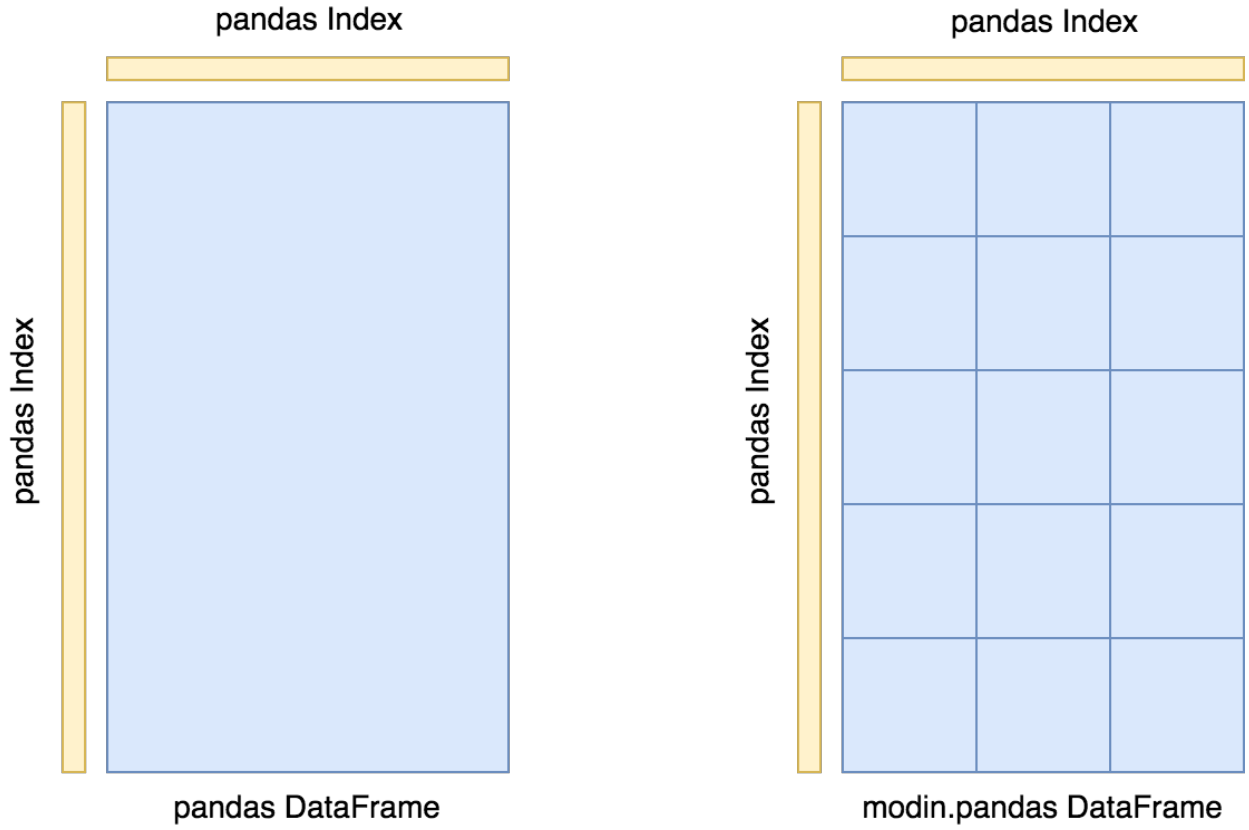
This is a list of execution engines and in-memory formats supported in Modin. If you would like to contribute a new execution engine or in-memory format, please see the documentation page on *contributing*.

- ***pandas on Ray***
  - Uses the Ray execution framework.
  - The storage format is *pandas* and the in-memory partition type is a pandas DataFrame.
  - For more information on the execution path, see the pandas on Ray page.
- ***pandas on Dask***
  - Uses the Dask Futures execution framework.
  - The storage format is *pandas* and the in-memory partition type is a pandas DataFrame.
  - For more information on the execution path, see the pandas on Dask page.
- ***pandas on Python***
  - Uses native python execution - mainly used for debugging.
  - The storage format is *pandas* and the in-memory partition type is a pandas DataFrame.
  - For more information on the execution path, see the pandas on Python page.
- **pandas on Ray (experimental)**
  - Uses the Ray execution framework.
  - The storage format is *pandas* and the in-memory partition type is a pandas DataFrame.
  - For more information on the execution path, see the experimental pandas on Ray page.
- ***OmniSci on Native*** **(experimental)**
  - Uses OmniSciDB as an engine.
  - The storage format is *omnisci* and the in-memory partition type is a pyarrow Table. When defaulting to pandas, the pandas DataFrame is used.
  - For more information on the execution path, see the OmniSci on Native page.
- ***Pyarrow on Ray*** **(experimental)**
  - Uses the Ray execution framework.
  - The storage format is *pyarrow* and the in-memory partition type is a pyarrow Table.
  - For more information on the execution path, see the Pyarrow on Ray page.
- **cuDF on Ray (experimental)**
  - Uses the Ray execution framework.

- The storage format is *cudf* and the in-memory partition type is a cuDF DataFrame.
- For more information on the execution path, see the cuDF on Ray page.

12.2.5 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. The following figure illustrates this concept.



Currently, the main in-memory format of each partition is a pandas DataFrame (pandas storage format). Omnisci, PyArrow and cuDF are also supported as experimental in-memory formats in Modin.

12.2.6 Index

We currently use the `pandas.Index` object for indexing both columns and rows. In the future, we will implement a distributed, pandas-compatible Index object in order to remove this scaling limitation from the system. Most workloads will not be affected by this scalability limit since it only appears when operating on more than 10's of billions of columns or rows. **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.

12.2.7 API

The API is the outer-most layer that faces users. The following classes contain Modin's implementation of the pandas API:

Base pandas Dataset API

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
```

(continues on next page)

(continued from previous page)

```
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
```

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)

Examples

```
>>> df = pd.DataFrame(
...     [[1, 2, 3, 4], [6, 7, 8, 9]], columns=["D", "B", "E", "A"], index=[1, 2]
... )
>>> other = pd.DataFrame(
...     [[10, 20, 30, 40], [60, 70, 80, 90], [600, 700, 800, 900]],
...     columns=["A", "B", "C", "D"],
...     index=[2, 3, 4],
... )
>>> df
   D  B  E  A
1  1  2  3  4
2  6  7  8  9
>>> other
    A    B    C    D
2   10   20   30   40
3   60   70   80   90
4  600  700  800  900
```

Align on columns:

```
>>> left, right = df.align(other, join="outer", axis=1)
>>> left
   A  B   C  D  E
1  4  2 NaN  1  3
2  9  7 NaN  6  8
>>> right
    A    B    C    D   E
2   10   20   30   40 NaN
3   60   70   80   90 NaN
4  600  700  800  900 NaN
```

We can also align on the index:

```
>>> left, right = df.align(other, join="outer", axis=0)
>>> left
    D    B    E    A
1  1.0  2.0  3.0  4.0
2  6.0  7.0  8.0  9.0
3  NaN  NaN  NaN  NaN
4  NaN  NaN  NaN  NaN
>>> right
    A      B      C      D
1   NaN    NaN    NaN    NaN
2  10.0   20.0   30.0   40.0
3  60.0   70.0   80.0   90.0
4  600.0  700.0  800.0  900.0
```

Finally, the default *axis=None* will align on both index and columns:

```
>>> left, right = df.align(other, join="outer", axis=None)
>>> left
     A    B   C    D    E
1  4.0  2.0 NaN  1.0  3.0
2  9.0  7.0 NaN  6.0  8.0
3  NaN  NaN NaN  NaN  NaN
4  NaN  NaN NaN  NaN  NaN
>>> right
       A      B      C      D   E
1    NaN    NaN    NaN    NaN NaN
2   10.0   20.0   30.0   40.0 NaN
3   60.0   70.0   80.0   90.0 NaN
4  600.0  700.0  800.0  900.0 NaN
```

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()
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*) –

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: bool_t | NoDefault = NoDefault.no_default*, *include_end: bool_t | NoDefault = NoDefault.no_default*, *inclusive: str | None = None*, *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.

  Deprecated since version 1.4.0: Arguments *include_start* and *include_end* have been deprecated to standardize boundary inputs. Use *inclusive* instead, to set each bound as closed or open.

- **include_end** (*bool, default True*) – Whether the end time needs to be included in the result.

  Deprecated since version 1.4.0: Arguments *include_start* and *include_end* have been deprecated to standardize boundary inputs. Use *inclusive* instead, to set each bound as closed or open.

- **inclusive** (*{"both", "neither", "left", "right"}, default "both"*) – Include boundaries; whether to set each bound as closed or open.
- **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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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
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
```

(continues on next page)

(continued from previous page)

```
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'))
```

(continues on next page)

(continued from previous page)

```
>>> 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
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
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(exclude=['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"),
```

(continues on next page)

(continued from previous page)

```
...   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
std      1.0
min      1.0
25%      1.5
```

(continues on next page)

(continued from previous page)

```
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
unique           3      NaN
top              f      NaN
freq             1      NaN
```

(continues on next page)

(continued from previous page)

```
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
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
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
```

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
           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. A tuple will be used as a single label and not treated as a list-like.
- **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
```

Drop a specific index combination from the MultiIndex DataFrame, i.e., drop the combination `'falcon'` and `'weight'`, which deletes only the corresponding row

```
>>> df.drop(index=('falcon', 'weight'))
                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
        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
```

(continues on next page)

(continued from previous page)

```
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
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** (*column label or sequence of labels, 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
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
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
```

(continues on next page)

(continued from previous page)

```
   B   True     True
   C   True     True
Q2 A  False     True
   B   True    False
   C   True    False
```

ewm(*com: float | None = None*, *span: float | None = None*, *halflife: float | TimedeltaConvertibleTypes | None = None*, *alpha: float | None = None*, *min_periods: int | None = 0*, *adjust: bool_t = True*, *ignore_na: bool_t = False*, *axis: Axis = 0*, *times: str | np.ndarray | BasePandasDataset | None = None*, *method: str = 'single'*) → ExponentialMovingWindow

Provide exponentially weighted (EW) calculations.

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

  $\alpha = 1 - \exp\left(-\ln(2)/halflife\right)$, 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 `np.nan`.

- **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:

  $$\begin{aligned} y_0 &= x_0 \\ y_t &= (1-\alpha)y_{t-1} + \alpha x_t, \end{aligned}$$

- **ignore_na** (*bool, default False*) – Ignore missing values when calculating weights.

  – 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`, 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*) – If `0` or `'index'`, calculate across the rows.

  If 1 or `'columns'`, calculate across the columns.

- **times** (*str, np.ndarray, Series, default None*) – New in version 1.1.0.

  Only applicable to `mean()`.

  Times corresponding to the observations. Must be monotonically increasing and `datetime64[ns]` dtype.

  If 1-D array like, a sequence with the same shape as the observations.

  Deprecated since version 1.4.0: If str, the name of the column in the DataFrame representing the times.

- **method** (*str {'single', 'table'}, default 'single'*) – New in version 1.4.0.

  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.

  Only applicable to `mean()`

Return type `ExponentialMovingWindow` subclass

See also:

rolling Provides rolling window calculations.

expanding Provides expanding transformations.

Notes

See pandas API documentation for pandas.DataFrame.ewm for more. See Windowing Operations for further usage details and examples.

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
```

(continues on next page)

(continued from previous page)

```
3  1.615385
4  3.670213
>>> df.ewm(alpha=2 / 3).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

adjust

```
>>> df.ewm(com=0.5, adjust=True).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
>>> df.ewm(com=0.5, adjust=False).mean()
          B
0  0.000000
1  0.666667
2  1.555556
3  1.555556
4  3.650794
```

ignore_na

```
>>> df.ewm(com=0.5, ignore_na=True).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.225000
>>> df.ewm(com=0.5, ignore_na=False).mean()
          B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

times

Exponentially weighted mean with weights calculated with a timedelta `halflife` relative to `times`.

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
1  0.585786
2  1.523889
3  1.523889
4  3.233686
```

expanding(*min_periods=1*, *center=None*, *axis=0*, *method='single'*)

Provide expanding window calculations.

Parameters

- **min_periods** (*int, default 1*) – Minimum number of observations in window required to have a value; otherwise, result is `np.nan`.
- **center** (*bool, default False*) – If False, set the window labels as the right edge of the window index.

  If True, set the window labels as the center of the window index.

  Deprecated since version 1.1.0.
- **axis** (*int or str, default 0*) – If `0` or `'index'`, roll across the rows.

  If `1` or `'columns'`, roll across the columns.
- **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.

Return type `Expanding` subclass

See also:

rolling Provides rolling window calculations.

ewm Provides exponential weighted functions.

Notes

See pandas API documentation for pandas.DataFrame.expanding for more. See Windowing Operations for further usage details and examples.

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
```

min_periods

Expanding sum with 1 vs 3 observations needed to calculate a value.

```
>>> df.expanding(1).sum()
     B
0  0.0
1  1.0
2  3.0
3  3.0
4  7.0
>>> df.expanding(3).sum()
     B
0  NaN
1  NaN
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.

Reference the user guide for more examples.

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.

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
```

(continues on next page)

(continued from previous page)

```
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
         one  two  three
rabbit    4    5      6
```

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 non-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
```

(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
```

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
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

Examples

```
>>> df = 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"),
... )
```

```
>>> df
            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
```

```
>>> df.get(["temp_celsius", "windspeed"])
            temp_celsius windspeed
2014-02-12          24.3      high
2014-02-13          31.0      high
2014-02-14          22.0    medium
2014-02-15          35.0    medium
```

If the key isn't found, the default value will be used.

```
>>> df.get(["temp_celsius", "temp_kelvin"], default="default_value")
'default_value'
```

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
```

(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
```

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
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
```

To check if `values` is *not* in the DataFrame, use the ~ operator:

```
>>> ~df.isin([0, 2])
        num_legs  num_wings
falcon     False      False
dog         True      False
```

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 other.

```
>>> other = pd.DataFrame({'num_legs': [8, 3], 'num_wings': [0, 2]},
...                      index=['spider', 'falcon'])
>>> df.isin(other)
        num_legs  num_wings
falcon     False       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: Axis | None | NoDefault = NoDefault.no_default*, *skipna=True*, *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.

Return type *Series* or *DataFrame* (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.kurt for more.

kurtosis(*axis: Axis | None | NoDefault = NoDefault.no_default*, *skipna=True*, *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.

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 non-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
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
```

(continues on next page)

(continued from previous page)

```
   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=True*, *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 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.

Return type *Series* or *DataFrame* (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.mad for more.

max(*axis: int | None | NoDefault = NoDefault.no_default*, *skipna=True*, *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.

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: int | None | NoDefault = NoDefault.no_default*, *skipna=True*, *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.

Return type *Series* or *DataFrame* (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.mean for more.

median(*axis: int | None | NoDefault = NoDefault.no_default*, *skipna=True*, *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.

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
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: int | None | NoDefault = NoDefault.no_default*, *skipna=True*, *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.

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
```

(continues on next page)

(continued from previous page)

```
         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
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
```

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
```

(continues on next page)

(continued from previous page)

```
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
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
```

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
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
```

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
```

(continues on next page)

(continued from previous page)

```
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.

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
          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 chainable functions that expect Series or DataFrames.

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.

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
           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: str = 'average'*, *numeric_only: bool_t | None | NoDefault = NoDefault.no_default*, *na_option: str = 'keep'*, *ascending: bool_t = True*, *pct: bool_t = 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
```

(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
```

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.

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']
```

(continues on next page)

(continued from previous page)

```
>>> 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
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]},
```

(continues on next page)

(continued from previous page)

```
...                     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
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")
>>> 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.

Return type *DataFrame*

Examples

```
>>> data = {
...     "class": ["Mammals", "Mammals", "Reptiles"],
...     "diet": ["Omnivore", "Carnivore", "Carnivore"],
...     "species": ["Humans", "Dogs", "Snakes"],
... }
>>> df = pd.DataFrame(data, columns=["class", "diet", "species"])
>>> df = df.set_index(["class", "diet"])
>>> df
                                  species
class      diet
Mammals    Omnivore                Humans
           Carnivore                 Dogs
Reptiles   Carnivore               Snakes
```

Let's reorder the levels of the index:

```
>>> df.reorder_levels(["diet", "class"])
                                  species
diet      class
Omnivore  Mammals                 Humans
Carnivore Mammals                   Dogs
          Reptiles                Snakes
```

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** (*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 string, must be one of the following:
  - '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
```

(continues on next page)

(continued from previous page)

```
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
2000-01-01 00:01:30    2
```

(continues on next page)

(continued from previous page)

```
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
2018-06    2.0
```

(continues on next page)

(continued from previous page)

```
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
2000-01-03 morning      14      50
```

(continues on next page)

(continued from previous page)

```
           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.

  If an integer, the fixed number of observations used for each window.

  If an offset, 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. To learn more about the offsets & frequency strings, please see this link.

  If a BaseIndexer subclass, 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 `np.nan`.

  For a window that is specified by an offset, `min_periods` will default to 1.

  For a window that is specified by an integer, `min_periods` will default to the size of the window.

- **center** (*bool, default False*) – If False, set the window labels as the right edge of the window index.

  If True, set the window labels as the center of the window index.

- **win_type** (*str, default None*) – If `None`, all points are evenly weighted.

  If a string, it must be a valid 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.

- **on** (*str, optional*) – For a DataFrame, a column label 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*) – If `0` or `'index'`, roll across the rows.

If 1 or `'columns'`, roll across the columns.

- **closed** (*str, default None*) – If `'right'`, the first point in the window is excluded from calculations.

  If `'left'`, the last point in the window is excluded from calculations.

  If `'both'`, the no points in the window are excluded from calculations.

  If `'neither'`, the first and last points in the window are excluded from calculations.

  Default `None` (`'right'`).

  Changed in version 1.2.0: The closed parameter with fixed windows is now supported.

- **method** (*str {'single', 'table'}, default 'single'*) – New in version 1.3.0.

  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.

Returns

- `Window` subclass if a `win_type` is passed
- `Rolling` subclass if `win_type` is not passed

See also:

expanding Provides expanding transformations.

ewm Provides exponential weighted functions.

Notes

See pandas API documentation for pandas.DataFrame.rolling for more. See Windowing Operations for further usage details and examples.

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
```

window

Rolling sum with a window length of 2 observations.

```
>>> df.rolling(2).sum()
     B
0  NaN
1  1.0
2  3.0
```

(continues on next page)

(continued from previous page)

```
3  NaN
4  NaN
```

Rolling sum with a window span of 2 seconds.

```
>>> df_time = 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_time
                       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
```

```
>>> df_time.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
```

Rolling sum with forward looking windows with 2 observations.

```
>>> 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
```

min_periods

Rolling sum with a window length of 2 observations, but only needs a minimum of 1 observation to calculate a value.

```
>>> df.rolling(2, min_periods=1).sum()
     B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

center

Rolling sum with the result assigned to the center of the window index.

```
>>> df.rolling(3, min_periods=1, center=True).sum()
     B
0  1.0
1  3.0
2  3.0
3  6.0
4  4.0
```

```
>>> df.rolling(3, min_periods=1, center=False).sum()
     B
0  0.0
1  1.0
2  3.0
3  3.0
4  6.0
```

win_type

Rolling sum with a window length of 2, using the Scipy `'gaussian'` window type. `std` is required in the aggregation function.

```
>>> df.rolling(2, win_type='gaussian').sum(std=3)
          B
0       NaN
1  0.986207
2  2.958621
3       NaN
4       NaN
```

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
  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
```

(continues on next page)

(continued from previous page)

```
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, np.random.Generator, optional*) – If int, array-like, or BitGenerator, seed for random number generator. If np.random.RandomState or np.random.Generator, use as given.

  Changed in version 1.1.0: array-like and BitGenerator object now passed to np.random.RandomState() as seed

  Changed in version 1.4.0: np.random.Generator objects now accepted

- **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=True*, *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.

Return type *Series* or *DataFrame* (if level specified)

Notes

See pandas API documentation for pandas.DataFrame.sem for more.

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
```

(continues on next page)

(continued from previous page)

```
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: int | None | NoDefault = NoDefault.no_default*, *skipna=True*, *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.

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
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
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
```

(continues on next page)

(continued from previous page)

```
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=True*, *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.

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*)

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                   ).set_index('person_id')
>>> df
           age  height
person_id
0           21    1.61
1           25    1.87
2           62    1.49
3           43    2.01
```

The standard deviation of the columns can be found as follows:

```
>>> df.std()
age       18.786076
height     0.237417
```

Alternatively, *ddof=0* can be set to normalize by N instead of N-1:

```
>>> df.std(ddof=0)
age       16.269219
height     0.205609
```

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
```

(continues on next page)

(continued from previous page)

```
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'],
```

(continues on next page)

(continued from previous page)

```
...                                    ['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 with levels swapped in MultiIndex.

Return type *DataFrame*

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
```

(continues on next page)

(continued from previous page)

```
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_csv.

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
- macOS : 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, path object, file-like object, or None, default None*) – String, path object (implementing os.PathLike[str]), or file-like object implementing a write() function. If None, 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'*) – For on-the-fly compression of the output data. If 'infer' and '%s' path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', or '.zst' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, or `zstandard.ZstdDecompressor`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

  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', 'bz2', 'zstd', and '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)
```

To write a csv file to a new folder or nested folder you will first need to create it using either Pathlib or os:

```
>>> from pathlib import Path
>>> filepath = Path('folder/subfolder/out.csv')
>>> filepath.parent.mkdir(parents=True, exist_ok=True)
>>> df.to_csv(filepath)
```

```
>>> import os
>>> os.makedirs('folder/subfolder', exist_ok=True)
>>> df.to_csv('folder/subfolder/out.csv')
```

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]}
- 'tight' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values], 'index_names' -> [index.names], 'column_names' -> [column.names]}
- 'records' : list like [{column -> value}, ... , {column -> value}]
- 'index' : dict like {index -> {column -> value}}

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

New in version 1.4.0: 'tight' as an allowed value for the `orient` argument

- **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}}
```

```
>>> df.to_dict('tight')
{'index': ['row1', 'row2'], 'columns': ['col1', 'col2'],
 'data': [[1, 0.5], [2, 0.75]], 'index_names': [None], 'column_names': [None]}
```

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}, default None*) – Specifies a compression level for data. A value of 0 or None 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
```

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, path object, file-like object, or None, default None*) – String, path object (implementing os.PathLike[str]), or file-like object implementing a write() function. If None, 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** (*str or dict, default 'infer'*) – For on-the-fly compression of the output data. If 'infer' and 'path_or_buf' path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', or '.zst' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, or `zstandard.ZstdDecompressor`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

  Changed in version 1.4.0: Zstandard support.

- **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)
{
    "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)
```

(continues on next page)

(continued from previous page)

```
[
    {
        "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"
    ],
```

(continues on next page)

(continued from previous page)

```
    [
        "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": [
            "index"
        ],
        "pandas_version": "1.4.0"
    },
    "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.

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:

Styler.to_latex Render a DataFrame to LaTeX with conditional formatting.

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

```
>>> df = pd.DataFrame(
...     data={"animal_1": ["elk", "pig"], "animal_2": ["dog", "quetzal"]}
... )
>>> print(df.to_markdown())
|    | animal_1   | animal_2   |
|---:|:-----------|:-----------|
|  0 | elk        | dog        |
|  1 | pig        | quetzal    |
```

Output markdown with a tabulate option.

```
>>> print(df.to_markdown(tablefmt="grid"))
+----+------------+------------+
|    | animal_1   | animal_2   |
+====+============+============+
|  0 | elk        | dog        |
+----+------------+------------+
```

(continues on next page)

(continued from previous page)

```
|  1 | pig        | 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.

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.

Return type DataFrame with PeriodIndex

Examples

```
>>> idx = pd.to_datetime(
...     [
...         "2001-03-31 00:00:00",
...         "2002-05-31 00:00:00",
...         "2003-08-31 00:00:00",
...     ]
... )
```

```
>>> idx
DatetimeIndex(['2001-03-31', '2002-05-31', '2003-08-31'],
dtype='datetime64[ns]', freq=None)
```

```
>>> idx.to_period("M")
PeriodIndex(['2001-03', '2002-05', '2003-08'], dtype='period[M]')
```

For the yearly frequency

```
>>> idx.to_period("Y")
PeriodIndex(['2001', '2002', '2003'], dtype='period[A-DEC]')
```

Notes

See pandas API documentation for pandas.DataFrame.to_period for more.

to_pickle(*path*, *compression: Optional[Union[Literal['infer', 'gzip', 'bz2', 'zip', 'xz', 'zstd'], Dict[str, Any]]] = 'infer'*, *protocol: int = 5*, *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** (*str or dict, default 'infer'*) – For on-the-fly compression of the output data. If 'infer' and 'path' path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', or '.zst' (otherwise no compression). Set to `None` for no compression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, or `zstandard.ZstdDecompressor`, respectively. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.
- **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
```

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.

Returns

Number of rows affected by to_sql. None is returned if the callable passed into `method` does not return the number of rows.

The number of returned rows affected is the sum of the `rowcount` attribute of `sqlite3.Cursor` or SQLAlchemy connectable which may not reflect the exact number of written rows as stipulated in the sqlite3 or SQLAlchemy.

New in version 1.4.0.

Return type None or int

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)
3
>>> 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')
2
```

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')
2
>>> 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')
2
>>> 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()})
3
```

```
>>> 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. If a list of ints is given every integers corresponds with one column. If a dict is given, the key references the column, while the value defines the space to use..
- **header** (*bool or sequence of str, 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.
- **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.
- **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_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.

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 the same axis shape 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
```

(continues on next page)

(continued from previous page)

```
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
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()
```

(continues on next page)

(continued from previous page)

```
                     A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
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.

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
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=True*, *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.

Return type *Series* or *DataFrame* (if level specified)

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                   ).set_index('person_id')
>>> df
           age  height
person_id
0           21    1.61
1           25    1.87
2           62    1.49
3           43    2.01
```

```
>>> df.var()
age       352.916667
height      0.056367
```

Alternatively, `ddof=0` can be set to normalize by N instead of N-1:

```
>>> df.var(ddof=0)
age       264.687500
height      0.042275
```

Notes

See pandas API documentation for pandas.DataFrame.var for more.

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)
```

(continues on next page)

(continued from previous page)

```
# 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)

# 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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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>]]

# 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.

12.2.8 Module/Class View

Modin's modules layout is shown below. Click on the links to deep dive into Modin's internal implementation details. The documentation covers most modules, with more docs being added everyday!

```
├──.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
```

12.3 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.

12.3.1 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.

12.3.2 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*.

12.3.3 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.

12.3.4 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).

12.3.5 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.

12.4 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')
```

12.5 pandas on Dask

This section describes usage related documents for the pandas on Dask 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 Dask. To enable the pandas on Dask execution you should set the following environment variables:

```
export MODIN_ENGINE=dask
export MODIN_STORAGE_FORMAT=pandas
```

or turn them on in source code:

```
import modin.config as cfg
cfg.Engine.put('dask')
cfg.StorageFormat.put('pandas')
```

12.6 pandas on Python

This section describes usage related documents for the pandas on Python component of Modin.

Modin uses pandas as the primary memory format of the underlying partitions and optimizes queries from the API layer in a specific way to this format. Since it is a default, you do not need to specify the pandas memory format, but we show how to explicitly set it below.

One of the execution engines that Modin uses is Python. This engine is sequential and used for debugging. To enable the pandas on Python execution you should set the following environment variables:

```
export MODIN_ENGINE=python
export MODIN_STORAGE_FORMAT=pandas
```

or turn a debug mode on:

```
export MODIN_DEBUG=True
export MODIN_STORAGE_FORMAT=pandas
```

or do the same in source code:

```
import modin.config as cfg
cfg.Engine.put('python')
cfg.StorageFormat.put('pandas')
```

```
import modin.config as cfg
cfg.IsDebug.put(True)
cfg.StorageFormat.put('pandas')
```

12.7 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 can set the following environment variable:

```
export MODIN_STORAGE_FORMAT=omnisci
```

or use it in your code:

```
import modin.config as cfg
cfg.StorageFormat.put('omnisci')
```

Since OmniSci is run through its native engine, Modin automatically sets `MODIN_ENGINE=Native` and you might not specify it explicitly. If for some reasons `Native` engine is explicitly set using `modin.config` or `MODIN_ENGINE` environment variable, make sure you also tell Modin that `Experimental` mode is turned on (`export MODIN_EXPERIMENTAL=true` or `cfg.IsExperimental.put(True)`) otherwise the following error occurs:

```
FactoryNotFoundError: Omnisci on Native is only accessible through the experimental API.
Run `import modin.experimental.pandas as pd` to use Omnisci on Native.
```

Note: If you encounter `LLVM ERROR: inconsistency in registered CommandLine options` error when using OmniSci, please refer to the respective section in Troubleshooting page to avoid the issue.

12.8 PyArrow on Ray

Coming Soon!

12.9 Modin SQL API

Modin's SQL API is currently a conceptual plan, Coming Soon!

12.9.1 Plans for future development

Our plans with the SQL API for Modin are to create an interface that allows you to intermix SQL and pandas operations without copying the entire dataset into a new structure between the two. This is possible due to the architecture of Modin. Currently, Modin has a query compiler that acts as an intermediate layer between the query language (e.g. SQL, pandas) and the execution (See *architecture* documentation for details).

We have implemented a simple example that can be found below. Feedback welcome!

```
>>> import modin.sql as sql
>>>
>>> conn = sql.connect("db_name")
>>> c = conn.cursor()
>>> c.execute("CREATE TABLE example (col1, col2, column 3, col4)")
>>> c.execute("INSERT INTO example VALUES ('1', 2.0, 'A String of information', True)")
  col1  col2                 column 3  col4
0    1   2.0  A String of information  True

>>> c.execute("INSERT INTO example VALUES ('6', 17.0, 'A String of different information
↪', False)")
  col1  col2                           column 3   col4
0    1   2.0            A String of information   True
1    6  17.0  A String of different information  False
```

CHAPTER

THIRTEEN

CONTACT

13.1 Slack

Join our Slack community to connect with Modin users and contributors, discuss, and ask questions about all things Modin-related.

13.2 Discussion forum

Check out our discussion forum to discuss release announcements, general questions, issues, use-cases, and tutorials.

13.3 Mailing List

General questions, potential contributors, and ideas can 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.

13.4 Issues

Bug reports and feature requests can be directed to the issues page of the Modin GitHub repo.

CHAPTER

FOURTEEN

SCALE YOUR PANDAS WORKFLOW BY CHANGING A SINGLE LINE OF CODE

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

FIFTEEN

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
```

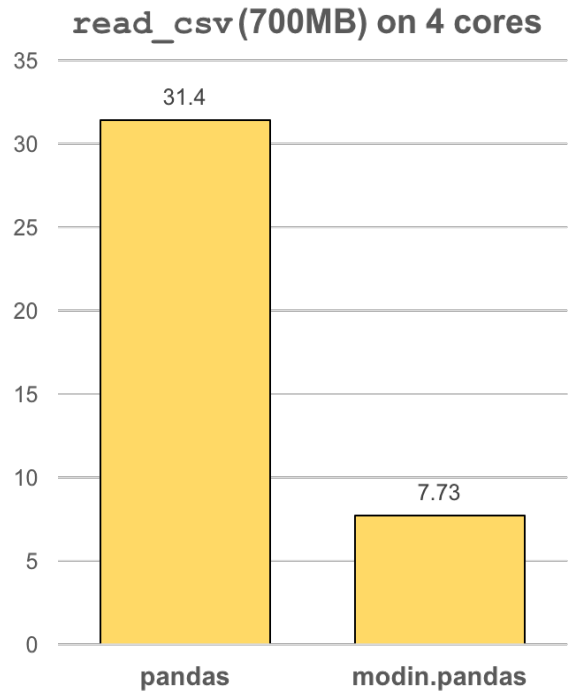
This can also be done within a notebook/interpreter before you import Modin:

```
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

SIXTEEN

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

SEVENTEEN

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 learning more about Modin, please check out the *Getting Started* guide then refer to the *Developer Documentation* section, where you can find system architecture, internal implementation details, and other useful information. Also check out the Github to view open issues and make contributions.

PYTHON MODULE INDEX

m

INDEX