pandas: powerful Python data analysis toolkit

Release 1.4.0

Wes McKinney and the Pandas Development Team

Jan 22, 2022

CONTENTS

Date: Jan 22, 2022 **Version**: 1.4.0

Download documentation: PDF Version | Zipped HTML

Previous versions: Documentation of previous pandas versions is available at pandas.pydata.org.

Useful links: Binary Installers | Source Repository | Issues & Ideas | Q&A Support | Mailing List

`pandas` is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

Getting started

New to *pandas*? Check out the getting started guides. They contain an introduction to *pandas'* main concepts and links to additional tutorials.

To the getting started guides

User guide

The user guide provides in-depth information on the key concepts of pandas with useful background information and explanation.

To the user guide

API reference

The reference guide contains a detailed description of the pandas API. The reference describes how the methods work and which parameters can be used. It assumes that you have an understanding of the key concepts.

To the reference guide

Developer guide

Saw a typo in the documentation? Want to improve existing functionalities? The contributing guidelines will guide you through the process of improving pandas.

To the development guide

CHAPTER

ONE

GETTING STARTED

1.1 Installation

Working with conda?

pandas is part of the Anaconda distribution and can be installed with Anaconda or Miniconda:

```
conda install pandas
```

Prefer pip?

pandas can be installed via pip from PyPI.

```
pip install pandas
```

In-depth instructions?

Installing a specific version? Installing from source? Check the advanced installation page.

Learn more

1.2 Intro to pandas

Straight to tutorial…

When working with tabular data, such as data stored in spreadsheets or databases, pandas is the right tool for you. pandas will help you to explore, clean, and process your data. In pandas, a data table is called a `DataFrame`.

To introduction tutorial

To user guide

Straight to tutorial…

pandas supports the integration with many file formats or data sources out of the box (csv, excel, sql, json, parquet,…). Importing data from each of these data sources is provided by function with the prefix `read_*`. Similarly, the `to_*` methods are used to store data.

To introduction tutorial

To user guide

Straight to tutorial…

Selecting or filtering specific rows and/or columns? Filtering the data on a condition? Methods for slicing, selecting, and extracting the data you need are available in pandas.

To introduction tutorial

To user guide

Straight to tutorial…

pandas provides plotting your data out of the box, using the power of Matplotlib. You can pick the plot type (scatter, bar, boxplot,…) corresponding to your data.

To introduction tutorial

To user guide

Straight to tutorial…

There is no need to loop over all rows of your data table to do calculations. Data manipulations on a column work elementwise. Adding a column to a `DataFrame` based on existing data in other columns is straightforward.

To introduction tutorial

To user guide

Straight to tutorial…

Basic statistics (mean, median, min, max, counts…) are easily calculable. These or custom aggregations can be applied on the entire data set, a sliding window of the data, or grouped by categories. The latter is also known as the split-apply-combine approach.

To introduction tutorial

To user guide

Straight to tutorial…

Change the structure of your data table in multiple ways. You can `melt()` your data table from wide to long/tidy form or `pivot()` from long to wide format. With aggregations built-in, a pivot table is created with a single command.

To introduction tutorial

To user guide

Straight to tutorial…

Multiple tables can be concatenated both column wise and row wise as database-like join/merge operations are provided to combine multiple tables of data.

To introduction tutorial

To user guide

Straight to tutorial…

pandas has great support for time series and has an extensive set of tools for working with dates, times, and time-indexed data.

To introduction tutorial

To user guide

Straight to tutorial…

Data sets do not only contain numerical data. pandas provides a wide range of functions to clean textual data and extract useful information from it.

To introduction tutorial

To user guide

1.3 Coming from…

Are you familiar with other software for manipulating tablular data? Learn the pandas-equivalent operations compared to software you already know:

The R programming language provides the `data.frame` data structure and multiple packages, such as tidyverse use and extend `data.frame` for convenient data handling functionalities similar to pandas.

Learn more

Already familiar to `SELECT`, `GROUP BY`, `JOIN`, etc.? Most of these SQL manipulations do have equivalents in pandas.

Learn more

The `data set` included in the STATA statistical software suite corresponds to the pandas `DataFrame`. Many of the operations known from STATA have an equivalent in pandas.

Learn more

Users of Excel or other spreadsheet programs will find that many of the concepts are transferrable to pandas.

Learn more

The SAS statistical software suite also provides the `data set` corresponding to the pandas `DataFrame`. Also SAS vectorized operations, filtering, string processing operations, and more have similar functions in pandas.

Learn more

1.4 Tutorials

For a quick overview of pandas functionality, see *10 Minutes to pandas*.

You can also reference the pandas cheat sheet for a succinct guide for manipulating data with pandas.

The community produces a wide variety of tutorials available online. Some of the material is enlisted in the community contributed *Community tutorials*.

1.4.1 Installation

The easiest way to install pandas is to install it as part of the Anaconda distribution, a cross platform distribution for data analysis and scientific computing. This is the recommended installation method for most users.

Instructions for installing from source, PyPI, ActivePython, various Linux distributions, or a development version are also provided.

Python version support

Officially Python 3.8, and 3.9.

Installing pandas

Installing with Anaconda

Installing pandas and the rest of the NumPy and SciPy stack can be a little difficult for inexperienced users.

The simplest way to install not only pandas, but Python and the most popular packages that make up the SciPy stack (IPython, NumPy, Matplotlib, …) is with Anaconda, a cross-platform (Linux, macOS, Windows) Python distribution for data analytics and scientific computing.

After running the installer, the user will have access to pandas and the rest of the SciPy stack without needing to install anything else, and without needing to wait for any software to be compiled.

Installation instructions for Anaconda can be found here.

A full list of the packages available as part of the Anaconda distribution can be found here.

Another advantage to installing Anaconda is that you don't need admin rights to install it. Anaconda can install in the user's home directory, which makes it trivial to delete Anaconda if you decide (just delete that folder).

Installing with Miniconda

The previous section outlined how to get pandas installed as part of the Anaconda distribution. However this approach means you will install well over one hundred packages and involves downloading the installer which is a few hundred megabytes in size.

If you want to have more control on which packages, or have a limited internet bandwidth, then installing pandas with Miniconda may be a better solution.

Conda is the package manager that the Anaconda distribution is built upon. It is a package manager that is both cross-platform and language agnostic (it can play a similar role to a pip and virtualenv combination).

Miniconda allows you to create a minimal self contained Python installation, and then use the Conda command to install additional packages.

First you will need Conda to be installed and downloading and running the Miniconda will do this for you. The installer can be found here

The next step is to create a new conda environment. A conda environment is like a virtualenv that allows you to specify a specific version of Python and set of libraries. Run the following commands from a terminal window:

```
conda create -n name_of_my_env python
```

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

```
source activate name_of_my_env
```

On Windows the command is:

```
activate name_of_my_env
```

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

```
conda install pandas
```

To install a specific pandas version:

```
conda install pandas=0.20.3
```

To install other packages, IPython for example:

```
conda install ipython
```

To install the full Anaconda distribution:

```
conda install anaconda
```

If you need packages that are available to pip but not conda, then install pip, and then use pip to install those packages:

```
conda install pip
pip install django
```

Installing from PyPI

pandas can be installed via pip from PyPI.

Note: You must have `pip>=19.3` to install from PyPI.

```
pip install pandas
```

Installing with ActivePython

Installation instructions for ActivePython can be found here. Versions 2.7, 3.5 and 3.6 include pandas.

Installing using your Linux distribution's package manager.

The commands in this table will install pandas for Python 3 from your distribution.

Distribution	Status	Download / Repository Link	Install method
Debian	stable	official Debian repository	`sudo apt-get install python3-pandas`
Debian & Ubuntu	unstable (latest packages)	NeuroDebian	`sudo apt-get install python3-pandas`
Ubuntu	stable	official Ubuntu repository	`sudo apt-get install python3-pandas`
OpenSuse	stable	OpenSuse Repository	`zypper in python3-pandas`
Fedora	stable	official Fedora repository	`dnf install python3-pandas`
Centos/RHEL	stable	EPEL repository	`yum install python3-pandas`

However, the packages in the linux package managers are often a few versions behind, so to get the newest version of pandas, it's recommended to install using the `pip` or `conda` methods described above.

Handling ImportErrors

If you encounter an ImportError, it usually means that Python couldn't find pandas in the list of available libraries. Python internally has a list of directories it searches through, to find packages. You can obtain these directories with:

```
import sys
sys.path
```

One way you could be encountering this error is if you have multiple Python installations on your system and you don't have pandas installed in the Python installation you're currently using. In Linux/Mac you can run `which python` on your terminal and it will tell you which Python installation you're using. If it's something like "/usr/bin/python", you're using the Python from the system, which is not recommended.

It is highly recommended to use `conda`, for quick installation and for package and dependency updates. You can find simple installation instructions for pandas in this document: installation instructions </getting_started.html>.

Installing from source

See the *contributing guide* for complete instructions on building from the git source tree. Further, see *creating a development environment* if you wish to create a pandas development environment.

Running the test suite

pandas is equipped with an exhaustive set of unit tests, covering about 97% of the code base as of this writing. To run it on your machine to verify that everything is working (and that you have all of the dependencies, soft and hard, installed), make sure you have pytest >= 6.0 and Hypothesis >= 3.58, then run:

```
>>> pd.test()
running: pytest --skip-slow --skip-network C:\Users\TP\Anaconda3\envs\py36\lib\site-
↪packages\pandas
```

(continues on next page)

(continued from previous page)

```
============================= test session starts ==============================
platform win32 -- Python 3.6.2, pytest-3.6.0, py-1.4.34, pluggy-0.4.0
rootdir: C:\Users\TP\Documents\Python\pandasdev\pandas, inifile: setup.cfg
collected 12145 items / 3 skipped

..................................................................S......
........S................................................................
.........................................................................

==================== 12130 passed, 12 skipped in 368.339 seconds =====================
```

Dependencies

Package	Minimum supported version
NumPy	1.18.5
python-dateutil	2.8.1
pytz	2020.1

Recommended dependencies

- numexpr: for accelerating certain numerical operations. `numexpr` uses multiple cores as well as smart chunking and caching to achieve large speedups. If installed, must be Version 2.7.1 or higher.
- bottleneck: for accelerating certain types of `nan` evaluations. `bottleneck` uses specialized cython routines to achieve large speedups. If installed, must be Version 1.3.1 or higher.

Note: You are highly encouraged to install these libraries, as they provide speed improvements, especially when working with large data sets.

Optional dependencies

pandas has many optional dependencies that are only used for specific methods. For example, `pandas.read_hdf()` requires the `pytables` package, while `DataFrame.to_markdown()` requires the `tabulate` package. If the optional dependency is not installed, pandas will raise an `ImportError` when the method requiring that dependency is called.

Visualization

Dependency	Minimum Version	Notes
matplotlib	3.3.2	Plotting library
Jinja2	2.11	Conditional formatting with DataFrame.style
tabulate	0.8.7	Printing in Markdown-friendly format (see tabulate)

Computation

Dependency	Minimum Version	Notes
SciPy	1.14.1	Miscellaneous statistical functions
numba	0.50.1	Alternative execution engine for rolling operations (see *Enhancing Performance*)
xarray	0.15.1	pandas-like API for N-dimensional data

Excel files

Dependency	Minimum Version	Notes
xlrd	2.0.1	Reading Excel
xlwt	1.3.0	Writing Excel
xlsxwriter	1.2.2	Writing Excel
openpyxl	3.0.3	Reading / writing for xlsx files
pyxlsb	1.0.6	Reading for xlsb files

HTML

Dependency	Minimum Version	Notes
BeautifulSoup4	4.8.2	HTML parser for read_html
html5lib	1.1	HTML parser for read_html
lxml	4.5.0	HTML parser for read_html

One of the following combinations of libraries is needed to use the top-level `read_html()` function:

- BeautifulSoup4 and html5lib
- BeautifulSoup4 and lxml
- BeautifulSoup4 and html5lib and lxml
- Only lxml, although see *HTML Table Parsing* for reasons as to why you should probably **not** take this approach.

Warning:

- if you install BeautifulSoup4 you must install either lxml or html5lib or both. `read_html()` will **not** work with *only* BeautifulSoup4 installed.
- You are highly encouraged to read *HTML Table Parsing gotchas*. It explains issues surrounding the installation and usage of the above three libraries.

XML

Dependency	Minimum Version	Notes
lxml	4.5.0	XML parser for read_xml and tree builder for to_xml

SQL databases

Dependency	Minimum Version	Notes
SQLAlchemy	1.4.0	SQL support for databases other than sqlite
psycopg2	2.8.4	PostgreSQL engine for sqlalchemy
pymysql	0.10.1	MySQL engine for sqlalchemy

Other data sources

Dependency	Minimum Version	Notes
PyTables	3.6.1	HDF5-based reading / writing
blosc	1.20.1	Compression for HDF5
zlib		Compression for HDF5
fastparquet	0.4.0	Parquet reading / writing
pyarrow	1.0.1	Parquet, ORC, and feather reading / writing
pyreadstat	1.1.0	SPSS files (.sav) reading

Warning:

- If you want to use `read_orc()`, it is highly recommended to install pyarrow using conda. The following is a summary of the environment in which `read_orc()` can work.

System	Conda	PyPI
Linux	Successful	Failed(pyarrow==3.0 Successful)
macOS	Successful	Failed
Windows	Failed	Failed

Access data in the cloud

Dependency	Minimum Version	Notes
fsspec	0.7.4	Handling files aside from simple local and HTTP
gcsfs	0.6.0	Google Cloud Storage access
pandas-gbq	0.14.0	Google Big Query access
s3fs	0.4.0	Amazon S3 access

Clipboard

Dependency	Minimum Version	Notes
PyQt4/PyQt5		Clipboard I/O
qtpy		Clipboard I/O
xclip		Clipboard I/O on linux
xsel		Clipboard I/O on linux

Compression

Dependency	Minimum Version	Notes
Zstandard		Zstandard compression

1.4.2 Package overview

pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, **real-world** data analysis in Python. Additionally, it has the broader goal of becoming **the most powerful and flexible open source data analysis/manipulation tool available in any language**. It is already well on its way toward this goal.

pandas is well suited for many different kinds of data:

- Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
- Ordered and unordered (not necessarily fixed-frequency) time series data.
- Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
- Any other form of observational / statistical data sets. The data need not be labeled at all to be placed into a pandas data structure

The two primary data structures of pandas, `Series` (1-dimensional) and `DataFrame` (2-dimensional), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering. For R users, `DataFrame` provides everything that R's `data.frame` provides and much more. pandas is built on top of NumPy and is intended to integrate well within a scientific computing environment with many other 3rd party libraries.

Here are just a few of the things that pandas does well:

- Easy handling of **missing data** (represented as NaN) in floating point as well as non-floating point data
- Size mutability: columns can be **inserted and deleted** from DataFrame and higher dimensional objects
- Automatic and explicit **data alignment**: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let `Series`, `DataFrame`, etc. automatically align the data for you in computations
- Powerful, flexible **group by** functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data
- Make it **easy to convert** ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects
- Intelligent label-based **slicing**, **fancy indexing**, and **subsetting** of large data sets
- Intuitive **merging** and **joining** data sets
- Flexible **reshaping** and pivoting of data sets

- **Hierarchical** labeling of axes (possible to have multiple labels per tick)
- Robust IO tools for loading data from **flat files** (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast **HDF5 format**
- **Time series**-specific functionality: date range generation and frequency conversion, moving window statistics, date shifting, and lagging.

Many of these principles are here to address the shortcomings frequently experienced using other languages / scientific research environments. For data scientists, working with data is typically divided into multiple stages: munging and cleaning data, analyzing / modeling it, then organizing the results of the analysis into a form suitable for plotting or tabular display. pandas is the ideal tool for all of these tasks.

Some other notes

- pandas is **fast**. Many of the low-level algorithmic bits have been extensively tweaked in Cython code. However, as with anything else generalization usually sacrifices performance. So if you focus on one feature for your application you may be able to create a faster specialized tool.
- pandas is a dependency of statsmodels, making it an important part of the statistical computing ecosystem in Python.
- pandas has been used extensively in production in financial applications.

Data structures

Dimensions	Name	Description
1	Series	1D labeled homogeneously-typed array
2	DataFrame	General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed column

Why more than one data structure?

The best way to think about the pandas data structures is as flexible containers for lower dimensional data. For example, DataFrame is a container for Series, and Series is a container for scalars. We would like to be able to insert and remove objects from these containers in a dictionary-like fashion.

Also, we would like sensible default behaviors for the common API functions which take into account the typical orientation of time series and cross-sectional data sets. When using the N-dimensional array (ndarrays) to store 2- and 3-dimensional data, a burden is placed on the user to consider the orientation of the data set when writing functions; axes are considered more or less equivalent (except when C- or Fortran-contiguousness matters for performance). In pandas, the axes are intended to lend more semantic meaning to the data; i.e., for a particular data set, there is likely to be a “right” way to orient the data. The goal, then, is to reduce the amount of mental effort required to code up data transformations in downstream functions.

For example, with tabular data (DataFrame) it is more semantically helpful to think of the **index** (the rows) and the **columns** rather than axis 0 and axis 1. Iterating through the columns of the DataFrame thus results in more readable code:

```python
for col in df.columns:
    series = df[col]
    # do something with series
```

Mutability and copying of data

All pandas data structures are value-mutable (the values they contain can be altered) but not always size-mutable. The length of a Series cannot be changed, but, for example, columns can be inserted into a DataFrame. However, the vast majority of methods produce new objects and leave the input data untouched. In general we like to **favor immutability** where sensible.

Getting support

The first stop for pandas issues and ideas is the Github Issue Tracker. If you have a general question, pandas community experts can answer through Stack Overflow.

Community

pandas is actively supported today by a community of like-minded individuals around the world who contribute their valuable time and energy to help make open source pandas possible. Thanks to all of our contributors.

If you're interested in contributing, please visit the *contributing guide*.

pandas is a NumFOCUS sponsored project. This will help ensure the success of the development of pandas as a world-class open-source project and makes it possible to donate to the project.

Project governance

The governance process that pandas project has used informally since its inception in 2008 is formalized in Project Governance documents. The documents clarify how decisions are made and how the various elements of our community interact, including the relationship between open source collaborative development and work that may be funded by for-profit or non-profit entities.

Wes McKinney is the Benevolent Dictator for Life (BDFL).

Development team

The list of the Core Team members and more detailed information can be found on the people's page of the governance repo.

Institutional partners

The information about current institutional partners can be found on pandas website page.

License

```
BSD 3-Clause License

Copyright (c) 2008-2011, AQR Capital Management, LLC, Lambda Foundry, Inc. and PyData␣
↪Development Team
All rights reserved.

Copyright (c) 2011-2021, Open source contributors.
```

(continues on next page)

(continued from previous page)

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

* Neither the name of the copyright holder nor the names of its
  contributors may be used to endorse or promote products derived from
  this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

1.4.3 Getting started tutorials

What kind of data does pandas handle?

I want to start using pandas

```
In [1]: import pandas as pd
```

To load the pandas package and start working with it, import the package. The community agreed alias for pandas is pd, so loading pandas as pd is assumed standard practice for all of the pandas documentation.

pandas data table representation

I want to store passenger data of the Titanic. For a number of passengers, I know the name (characters), age (integers) and sex (male/female) data.

```
In [2]: df = pd.DataFrame(
   ...:     {
   ...:         "Name": [
   ...:             "Braund, Mr. Owen Harris",
   ...:             "Allen, Mr. William Henry",
   ...:             "Bonnell, Miss. Elizabeth",
   ...:         ],
```

(continues on next page)

(continued from previous page)

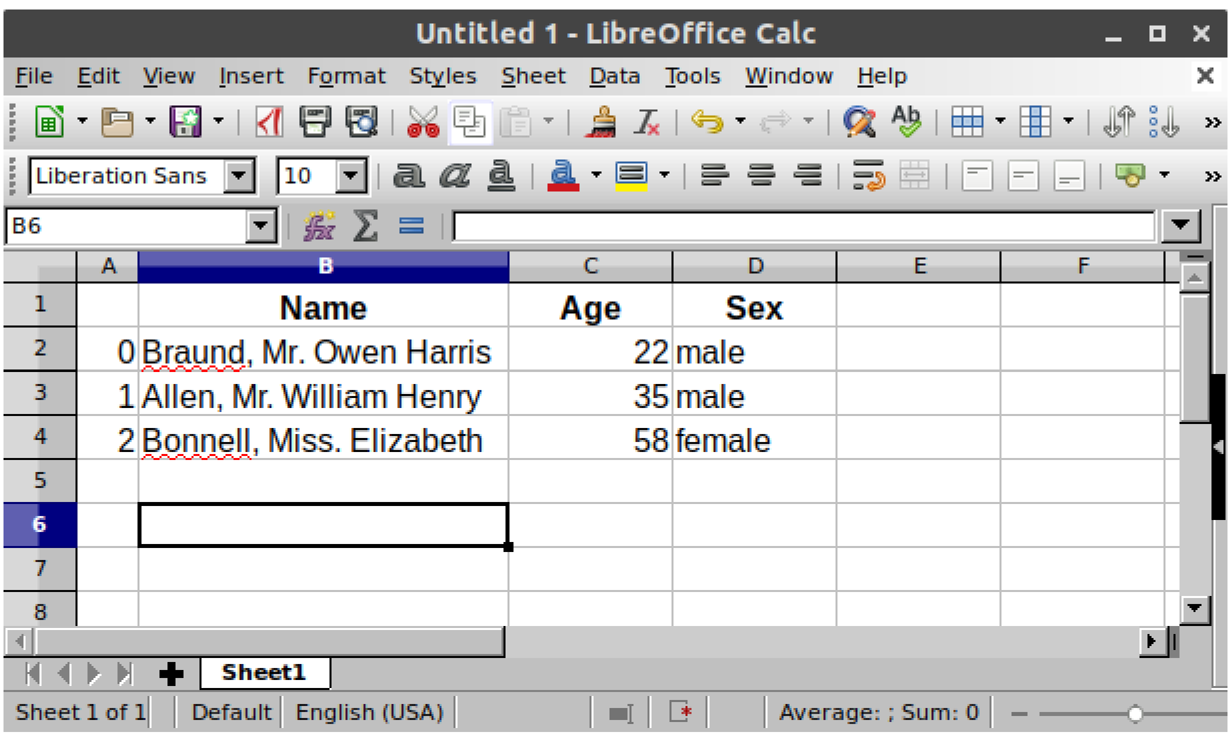
```
   ...:         "Age": [22, 35, 58],
   ...:         "Sex": ["male", "male", "female"],
   ...:     }
   ...: )
   ...:

In [3]: df
Out[3]:
                       Name  Age     Sex
0   Braund, Mr. Owen Harris   22    male
1  Allen, Mr. William Henry   35    male
2  Bonnell, Miss. Elizabeth   58  female
```

To manually store data in a table, create a `DataFrame`. When using a Python dictionary of lists, the dictionary keys will be used as column headers and the values in each list as columns of the `DataFrame`.

A *`DataFrame`* is a 2-dimensional data structure that can store data of different types (including characters, integers, floating point values, categorical data and more) in columns. It is similar to a spreadsheet, a SQL table or the `data.frame` in R.

- The table has 3 columns, each of them with a column label. The column labels are respectively `Name`, `Age` and `Sex`.
- The column `Name` consists of textual data with each value a string, the column `Age` are numbers and the column `Sex` is textual data.

In spreadsheet software, the table representation of our data would look very similar:



Each column in a DataFrame is a Series

I'm just interested in working with the data in the column Age

```
In [4]: df["Age"]
Out[4]:
0    22
1    35
2    58
Name: Age, dtype: int64
```

When selecting a single column of a pandas *DataFrame*, the result is a pandas *Series*. To select the column, use the column label in between square brackets [].

Note: If you are familiar to Python dictionaries, the selection of a single column is very similar to selection of dictionary values based on the key.

You can create a Series from scratch as well:

```
In [5]: ages = pd.Series([22, 35, 58], name="Age")

In [6]: ages
Out[6]:
0    22
1    35
2    58
Name: Age, dtype: int64
```

A pandas Series has no column labels, as it is just a single column of a DataFrame. A Series does have row labels.

Do something with a DataFrame or Series

I want to know the maximum Age of the passengers

We can do this on the DataFrame by selecting the Age column and applying max():

```
In [7]: df["Age"].max()
Out[7]: 58
```

Or to the Series:

```
In [8]: ages.max()
Out[8]: 58
```

As illustrated by the max() method, you can *do* things with a DataFrame or Series. pandas provides a lot of functionalities, each of them a *method* you can apply to a DataFrame or Series. As methods are functions, do not forget to use parentheses ().

I'm interested in some basic statistics of the numerical data of my data table

```
In [9]: df.describe()
Out[9]:
             Age
count   3.000000
mean   38.333333
std    18.230012
min    22.000000
25%    28.500000
50%    35.000000
75%    46.500000
max    58.000000
```

The *describe()* method provides a quick overview of the numerical data in a `DataFrame`. As the `Name` and `Sex` columns are textual data, these are by default not taken into account by the *describe()* method.

Many pandas operations return a `DataFrame` or a `Series`. The *describe()* method is an example of a pandas operation returning a pandas `Series` or a pandas `DataFrame`.

Check more options on `describe` in the user guide section about *aggregations with describe*

Note: This is just a starting point. Similar to spreadsheet software, pandas represents data as a table with columns and rows. Apart from the representation, also the data manipulations and calculations you would do in spreadsheet software are supported by pandas. Continue reading the next tutorials to get started!

- Import the package, aka `import pandas as pd`
- A table of data is stored as a pandas `DataFrame`
- Each column in a `DataFrame` is a `Series`
- You can do things by applying a method to a `DataFrame` or `Series`

A more extended explanation to `DataFrame` and `Series` is provided in the *introduction to data structures*.

```
In [1]: import pandas as pd
```

This tutorial uses the Titanic data set, stored as CSV. The data consists of the following data columns:

- PassengerId: Id of every passenger.
- Survived: This feature have value 0 and 1. 0 for not survived and 1 for survived.
- Pclass: There are 3 classes: Class 1, Class 2 and Class 3.
- Name: Name of passenger.
- Sex: Gender of passenger.
- Age: Age of passenger.
- SibSp: Indication that passenger have siblings and spouse.
- Parch: Whether a passenger is alone or have family.
- Ticket: Ticket number of passenger.
- Fare: Indicating the fare.
- Cabin: The cabin of passenger.
- Embarked: The embarked category.

How do I read and write tabular data?

I want to analyze the Titanic passenger data, available as a CSV file.

```
In [2]: titanic = pd.read_csv("data/titanic.csv")
```

pandas provides the *read_csv()* function to read data stored as a csv file into a pandas `DataFrame`. pandas supports many different file formats or data sources out of the box (csv, excel, sql, json, parquet, …), each of them with the prefix `read_*`.

Make sure to always have a check on the data after reading in the data. When displaying a `DataFrame`, the first and last 5 rows will be shown by default:

```
In [3]: titanic
Out[3]:
     PassengerId  Survived  Pclass                                               Name  ␣
↪ Sex  ...  Parch            Ticket     Fare Cabin  Embarked
0              1         0       3                            Braund, Mr. Owen Harris  ␣
↪male  ...      0         A/5 21171   7.2500   NaN         S
1              2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th... ␣
↪female  ...      0          PC 17599  71.2833   C85         C
2              3         1       3                             Heikkinen, Miss. Laina ␣
↪female  ...      0  STON/O2. 3101282   7.9250   NaN         S
3              4         1       1       Futrelle, Mrs. Jacques Heath (Lily May Peel) ␣
↪female  ...      0            113803  53.1000  C123         S
4              5         0       3                           Allen, Mr. William Henry  ␣
↪male  ...      0            373450   8.0500   NaN         S
..           ...       ...     ...                                                ...  ␣
↪ ...  ...    ...               ...      ...   ...       ...
886          887         0       2                              Montvila, Rev. Juozas  ␣
↪male  ...      0            211536  13.0000   NaN         S
887          888         1       1                       Graham, Miss. Margaret Edith ␣
↪female  ...      0            112053  30.0000   B42         S
888          889         0       3           Johnston, Miss. Catherine Helen "Carrie" ␣
↪female  ...      2        W./C. 6607  23.4500   NaN         S
889          890         1       1                              Behr, Mr. Karl Howell  ␣
↪male  ...      0            111369  30.0000  C148         C
890          891         0       3                                Dooley, Mr. Patrick  ␣
↪male  ...      0            370376   7.7500   NaN         Q

[891 rows x 12 columns]
```

I want to see the first 8 rows of a pandas DataFrame.

```
In [4]: titanic.head(8)
Out[4]:
   PassengerId  Survived  Pclass                                               Name  ␣
↪Sex  ...  Parch            Ticket     Fare Cabin  Embarked
0            1         0       3                            Braund, Mr. Owen Harris  ␣
↪male  ...      0         A/5 21171   7.2500   NaN         S
1            2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th... ␣
↪female  ...      0          PC 17599  71.2833   C85         C
2            3         1       3                             Heikkinen, Miss. Laina ␣
↪female  ...      0  STON/O2. 3101282   7.9250   NaN         S
```

(continues on next page)

(continued from previous page)

```
3           4         1       1       Futrelle, Mrs. Jacques Heath (Lily May Peel) ␣
↪female   ...      0            113803  53.1000  C123          S
4           5         0       3                           Allen, Mr. William Henry  ␣
↪male   ...      0            373450   8.0500   NaN          S
5           6         0       3                                   Moran, Mr. James  ␣
↪male   ...      0            330877   8.4583   NaN          Q
6           7         0       1                            McCarthy, Mr. Timothy J  ␣
↪male   ...      0             17463  51.8625   E46          S
7           8         0       3                     Palsson, Master. Gosta Leonard  ␣
↪male   ...      1            349909  21.0750   NaN          S

[8 rows x 12 columns]
```

To see the first N rows of a `DataFrame`, use the *head()* method with the required number of rows (in this case 8) as argument.

Note: Interested in the last N rows instead? pandas also provides a *tail()* method. For example, `titanic.tail(10)` will return the last 10 rows of the DataFrame.

A check on how pandas interpreted each of the column data types can be done by requesting the pandas `dtypes` attribute:

```
In [5]: titanic.dtypes
Out[5]:
PassengerId      int64
Survived         int64
Pclass           int64
Name            object
Sex             object
Age            float64
SibSp            int64
Parch            int64
Ticket          object
Fare           float64
Cabin           object
Embarked        object
dtype: object
```

For each of the columns, the used data type is enlisted. The data types in this `DataFrame` are integers (`int64`), floats (`float64`) and strings (`object`).

Note: When asking for the `dtypes`, no brackets are used! `dtypes` is an attribute of a `DataFrame` and `Series`. Attributes of `DataFrame` or `Series` do not need brackets. Attributes represent a characteristic of a `DataFrame`/`Series`, whereas a method (which requires brackets) *do* something with the `DataFrame`/`Series` as introduced in the *first tutorial*.

My colleague requested the Titanic data as a spreadsheet.

```
In [6]: titanic.to_excel("titanic.xlsx", sheet_name="passengers", index=False)
```

Whereas `read_*` functions are used to read data to pandas, the `to_*` methods are used to store data. The *to_excel()*

method stores the data as an excel file. In the example here, the `sheet_name` is named *passengers* instead of the default *Sheet1*. By setting `index=False` the row index labels are not saved in the spreadsheet.

The equivalent read function `read_excel()` will reload the data to a `DataFrame`:

```
In [7]: titanic = pd.read_excel("titanic.xlsx", sheet_name="passengers")
```

```
In [8]: titanic.head()
Out[8]:
   PassengerId  Survived  Pclass                                               Name   ␣
↪Sex  ...  Parch            Ticket     Fare Cabin  Embarked
0            1         0       3                            Braund, Mr. Owen Harris   ␣
↪male  ...      0         A/5 21171   7.2500   NaN         S
1            2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th... ␣
↪female  ...      0          PC 17599  71.2833   C85         C
2            3         1       3                             Heikkinen, Miss. Laina ␣
↪female  ...      0  STON/O2. 3101282   7.9250   NaN         S
3            4         1       1       Futrelle, Mrs. Jacques Heath (Lily May Peel) ␣
↪female  ...      0            113803  53.1000  C123         S
4            5         0       3                           Allen, Mr. William Henry   ␣
↪male  ...      0            373450   8.0500   NaN         S

[5 rows x 12 columns]
```

I'm interested in a technical summary of a `DataFrame`

```
In [9]: titanic.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

The method *`info()`* provides technical information about a `DataFrame`, so let's explain the output in more detail:

- It is indeed a *`DataFrame`*.
- There are 891 entries, i.e. 891 rows.
- Each row has a row label (aka the `index`) with values ranging from 0 to 890.
- The table has 12 columns. Most columns have a value for each of the rows (all 891 values are `non-null`). Some columns do have missing values and less than 891 `non-null` values.

- The columns Name, Sex, Cabin and Embarked consists of textual data (strings, aka object). The other columns are numerical data with some of them whole numbers (aka integer) and others are real numbers (aka float).
- The kind of data (characters, integers,...) in the different columns are summarized by listing the dtypes.
- The approximate amount of RAM used to hold the DataFrame is provided as well.
- Getting data in to pandas from many different file formats or data sources is supported by read_* functions.
- Exporting data out of pandas is provided by different to_*methods.
- The head/tail/info methods and the dtypes attribute are convenient for a first check.

For a complete overview of the input and output possibilities from and to pandas, see the user guide section about *reader and writer functions*.

```
In [1]: import pandas as pd
```

This tutorial uses the Titanic data set, stored as CSV. The data consists of the following data columns:

- PassengerId: Id of every passenger.
- Survived: This feature have value 0 and 1. 0 for not survived and 1 for survived.
- Pclass: There are 3 classes: Class 1, Class 2 and Class 3.
- Name: Name of passenger.
- Sex: Gender of passenger.
- Age: Age of passenger.
- SibSp: Indication that passenger have siblings and spouse.
- Parch: Whether a passenger is alone or have family.
- Ticket: Ticket number of passenger.
- Fare: Indicating the fare.
- Cabin: The cabin of passenger.
- Embarked: The embarked category.

```
In [2]: titanic = pd.read_csv("data/titanic.csv")

In [3]: titanic.head()
Out[3]:
   PassengerId  Survived  Pclass                                               Name  ␣
↪Sex  ...  Parch            Ticket     Fare Cabin  Embarked
0            1         0       3                            Braund, Mr. Owen Harris  ␣
↪male  ...      0         A/5 21171   7.2500   NaN         S
1            2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th... ␣
↪female  ...      0          PC 17599  71.2833   C85         C
2            3         1       3                             Heikkinen, Miss. Laina ␣
↪female  ...      0  STON/O2. 3101282   7.9250   NaN         S
3            4         1       1       Futrelle, Mrs. Jacques Heath (Lily May Peel) ␣
↪female  ...      0            113803  53.1000  C123         S
4            5         0       3                           Allen, Mr. William Henry  ␣
↪male  ...      0            373450   8.0500   NaN         S

[5 rows x 12 columns]
```

How do I select a subset of a DataFrame?

How do I select specific columns from a DataFrame?

I'm interested in the age of the Titanic passengers.

```
In [4]: ages = titanic["Age"]

In [5]: ages.head()
Out[5]:
0    22.0
1    38.0
2    26.0
3    35.0
4    35.0
Name: Age, dtype: float64
```

To select a single column, use square brackets `[]` with the column name of the column of interest.

Each column in a *DataFrame* is a *Series*. As a single column is selected, the returned object is a pandas *Series*. We can verify this by checking the type of the output:

```
In [6]: type(titanic["Age"])
Out[6]: pandas.core.series.Series
```

And have a look at the `shape` of the output:

```
In [7]: titanic["Age"].shape
Out[7]: (891,)
```

DataFrame.shape is an attribute (remember *tutorial on reading and writing*, do not use parentheses for attributes) of a pandas `Series` and `DataFrame` containing the number of rows and columns: *(nrows, ncolumns)*. A pandas Series is 1-dimensional and only the number of rows is returned.

I'm interested in the age and sex of the Titanic passengers.

```
In [8]: age_sex = titanic[["Age", "Sex"]]

In [9]: age_sex.head()
Out[9]:
    Age     Sex
0  22.0    male
1  38.0  female
2  26.0  female
3  35.0  female
4  35.0    male
```

To select multiple columns, use a list of column names within the selection brackets `[]`.

Note: The inner square brackets define a Python list with column names, whereas the outer brackets are used to select the data from a pandas `DataFrame` as seen in the previous example.

The returned data type is a pandas DataFrame:

```
In [10]: type(titanic[["Age", "Sex"]])
Out[10]: pandas.core.frame.DataFrame
```

```
In [11]: titanic[["Age", "Sex"]].shape
Out[11]: (891, 2)
```

The selection returned a `DataFrame` with 891 rows and 2 columns. Remember, a `DataFrame` is 2-dimensional with both a row and column dimension.

For basic information on indexing, see the user guide section on *indexing and selecting data*.

How do I filter specific rows from a `DataFrame`?

I'm interested in the passengers older than 35 years.

```
In [12]: above_35 = titanic[titanic["Age"] > 35]

In [13]: above_35.head()
Out[13]:
    PassengerId  Survived  Pclass                                               Name  ␣
↪Sex  ...  Parch    Ticket     Fare Cabin  Embarked
1             2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th... ␣
↪female  ...      0  PC 17599  71.2833   C85         C
6             7         0       1                            McCarthy, Mr. Timothy J  ␣
↪male  ...      0     17463  51.8625   E46         S
11           12         1       1                           Bonnell, Miss. Elizabeth ␣
↪female  ...      0    113783  26.5500  C103         S
13           14         0       3                        Andersson, Mr. Anders Johan  ␣
↪male  ...      5    347082  31.2750   NaN         S
15           16         1       2                   Hewlett, Mrs. (Mary D Kingcome) ␣
↪female  ...      0    248706  16.0000   NaN         S

[5 rows x 12 columns]
```

To select rows based on a conditional expression, use a condition inside the selection brackets `[]`.

The condition inside the selection brackets `titanic["Age"] > 35` checks for which rows the `Age` column has a value larger than 35:

```
In [14]: titanic["Age"] > 35
Out[14]:
0      False
1       True
2      False
3      False
4      False
       ...
886    False
887    False
888    False
889    False
```

(continues on next page)

(continued from previous page)

```
890    False
Name: Age, Length: 891, dtype: bool
```

The output of the conditional expression (>, but also ==, !=, <, <=,… would work) is actually a pandas `Series` of boolean values (either `True` or `False`) with the same number of rows as the original `DataFrame`. Such a `Series` of boolean values can be used to filter the `DataFrame` by putting it in between the selection brackets `[]`. Only rows for which the value is `True` will be selected.

We know from before that the original Titanic `DataFrame` consists of 891 rows. Let's have a look at the number of rows which satisfy the condition by checking the `shape` attribute of the resulting `DataFrame` `above_35`:

```
In [15]: above_35.shape
Out[15]: (217, 12)
```

I'm interested in the Titanic passengers from cabin class 2 and 3.

```
In [16]: class_23 = titanic[titanic["Pclass"].isin([2, 3])]

In [17]: class_23.head()
Out[17]: 
   PassengerId  Survived  Pclass                            Name     Sex   Age  SibSp ␣
↪Parch            Ticket     Fare Cabin Embarked
0            1         0       3         Braund, Mr. Owen Harris    male  22.0      1    ␣
↪   0         A/5 21171   7.2500   NaN        S
2            3         1       3          Heikkinen, Miss. Laina  female  26.0      0    ␣
↪   0  STON/O2. 3101282   7.9250   NaN        S
4            5         0       3        Allen, Mr. William Henry    male  35.0      0    ␣
↪   0            373450   8.0500   NaN        S
5            6         0       3                Moran, Mr. James    male   NaN      0    ␣
↪   0            330877   8.4583   NaN        Q
7            8         0       3  Palsson, Master. Gosta Leonard    male   2.0      3    ␣
↪   1            349909  21.0750   NaN        S
```

Similar to the conditional expression, the *isin()* conditional function returns a `True` for each row the values are in the provided list. To filter the rows based on such a function, use the conditional function inside the selection brackets `[]`. In this case, the condition inside the selection brackets `titanic["Pclass"].isin([2, 3])` checks for which rows the `Pclass` column is either 2 or 3.

The above is equivalent to filtering by rows for which the class is either 2 or 3 and combining the two statements with an `|` (or) operator:

```
In [18]: class_23 = titanic[(titanic["Pclass"] == 2) | (titanic["Pclass"] == 3)]

In [19]: class_23.head()
Out[19]: 
   PassengerId  Survived  Pclass                            Name     Sex   Age  SibSp ␣
↪Parch            Ticket     Fare Cabin Embarked
0            1         0       3         Braund, Mr. Owen Harris    male  22.0      1    ␣
↪   0         A/5 21171   7.2500   NaN        S
2            3         1       3          Heikkinen, Miss. Laina  female  26.0      0    ␣
↪   0  STON/O2. 3101282   7.9250   NaN        S
4            5         0       3        Allen, Mr. William Henry    male  35.0      0    ␣
↪   0            373450   8.0500   NaN        S
5            6         0       3                Moran, Mr. James    male   NaN      0    ␣
↪   0            330877   8.4583   NaN        Q
```

(continues on next page)

(continued from previous page)

```
7            8         0       3  Palsson, Master. Gosta Leonard    male   2.0      3   ␣
↪  1            349909  21.0750   NaN        S
```

Note: When combining multiple conditional statements, each condition must be surrounded by parentheses `()`. Moreover, you can not use `or`/`and` but need to use the `or` operator `|` and the `and` operator `&`.

See the dedicated section in the user guide about *boolean indexing* or about the *isin function*.

I want to work with passenger data for which the age is known.

```
In [20]: age_no_na = titanic[titanic["Age"].notna()]

In [21]: age_no_na.head()
Out[21]:
   PassengerId  Survived  Pclass                                               Name   ␣
↪Sex   ...  Parch            Ticket     Fare Cabin  Embarked
0            1         0       3                            Braund, Mr. Owen Harris   ␣
↪male  ...      0         A/5 21171   7.2500   NaN         S
1            2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th... ␣
↪female  ...      0          PC 17599  71.2833   C85         C
2            3         1       3                             Heikkinen, Miss. Laina ␣
↪female  ...      0  STON/O2. 3101282   7.9250   NaN         S
3            4         1       1       Futrelle, Mrs. Jacques Heath (Lily May Peel) ␣
↪female  ...      0            113803  53.1000  C123         S
4            5         0       3                           Allen, Mr. William Henry   ␣
↪male  ...      0            373450   8.0500   NaN         S

[5 rows x 12 columns]
```

The *`notna()`* conditional function returns a `True` for each row the values are not an `Null` value. As such, this can be combined with the selection brackets `[]` to filter the data table.

You might wonder what actually changed, as the first 5 lines are still the same values. One way to verify is to check if the shape has changed:

```
In [22]: age_no_na.shape
Out[22]: (714, 12)
```

For more dedicated functions on missing values, see the user guide section about *handling missing data*.

How do I select specific rows and columns from a `DataFrame`?

I'm interested in the names of the passengers older than 35 years.

```
In [23]: adult_names = titanic.loc[titanic["Age"] > 35, "Name"]

In [24]: adult_names.head()
Out[24]:
1      Cumings, Mrs. John Bradley (Florence Briggs Th...
```

(continues on next page)

(continued from previous page)

```
6                                McCarthy, Mr. Timothy J
11                              Bonnell, Miss. Elizabeth
13                           Andersson, Mr. Anders Johan
15                      Hewlett, Mrs. (Mary D Kingcome)
Name: Name, dtype: object
```

In this case, a subset of both rows and columns is made in one go and just using selection brackets [] is not sufficient anymore. The loc/iloc operators are required in front of the selection brackets []. When using loc/iloc, the part before the comma is the rows you want, and the part after the comma is the columns you want to select.

When using the column names, row labels or a condition expression, use the loc operator in front of the selection brackets []. For both the part before and after the comma, you can use a single label, a list of labels, a slice of labels, a conditional expression or a colon. Using a colon specifies you want to select all rows or columns.

I'm interested in rows 10 till 25 and columns 3 to 5.

```
In [25]: titanic.iloc[9:25, 2:5]
Out[25]:
    Pclass                                 Name     Sex
9        2  Nasser, Mrs. Nicholas (Adele Achem)  female
10       3      Sandstrom, Miss. Marguerite Rut  female
11       1             Bonnell, Miss. Elizabeth  female
12       3       Saundercock, Mr. William Henry    male
13       3          Andersson, Mr. Anders Johan    male
..     ...                                  ...     ...
20       2                 Fynney, Mr. Joseph J    male
21       2                Beesley, Mr. Lawrence    male
22       3          McGowan, Miss. Anna "Annie"  female
23       1         Sloper, Mr. William Thompson    male
24       3        Palsson, Miss. Torborg Danira  female

[16 rows x 3 columns]
```

Again, a subset of both rows and columns is made in one go and just using selection brackets [] is not sufficient anymore. When specifically interested in certain rows and/or columns based on their position in the table, use the iloc operator in front of the selection brackets [].

When selecting specific rows and/or columns with loc or iloc, new values can be assigned to the selected data. For example, to assign the name anonymous to the first 3 elements of the third column:

```
In [26]: titanic.iloc[0:3, 3] = "anonymous"

In [27]: titanic.head()
Out[27]:
   PassengerId  Survived  Pclass                                          Name     Sex  .
↪..  Parch            Ticket     Fare Cabin  Embarked
0            1         0       3                                     anonymous    male  .
↪..      0         A/5 21171   7.2500   NaN         S
1            2         1       1                                     anonymous  female  .
↪..      0          PC 17599  71.2833   C85         C
2            3         1       3                                     anonymous  female  .
↪..      0  STON/O2. 3101282   7.9250   NaN         S
3            4         1       1  Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  .
↪..      0            113803  53.1000  C123         S
```

(continues on next page)

(continued from previous page)

```
4           5          0       3                    Allen, Mr. William Henry    male  .
↪..      0             373450   8.0500   NaN          S

[5 rows x 12 columns]
```

See the user guide section on *different choices for indexing* to get more insight in the usage of `loc` and `iloc`.

- When selecting subsets of data, square brackets `[]` are used.
- Inside these brackets, you can use a single column/row label, a list of column/row labels, a slice of labels, a conditional expression or a colon.
- Select specific rows and/or columns using `loc` when using the row and column names
- Select specific rows and/or columns using `iloc` when using the positions in the table
- You can assign new values to a selection based on `loc`/`iloc`.

A full overview of indexing is provided in the user guide pages on *indexing and selecting data*.

```
In [1]: import pandas as pd

In [2]: import matplotlib.pyplot as plt
```

For this tutorial, air quality data about NO_2 is used, made available by openaq and using the py-openaq package. The `air_quality_no2.csv` data set provides NO_2 values for the measurement stations *FR04014*, *BETR801* and *London Westminster* in respectively Paris, Antwerp and London.

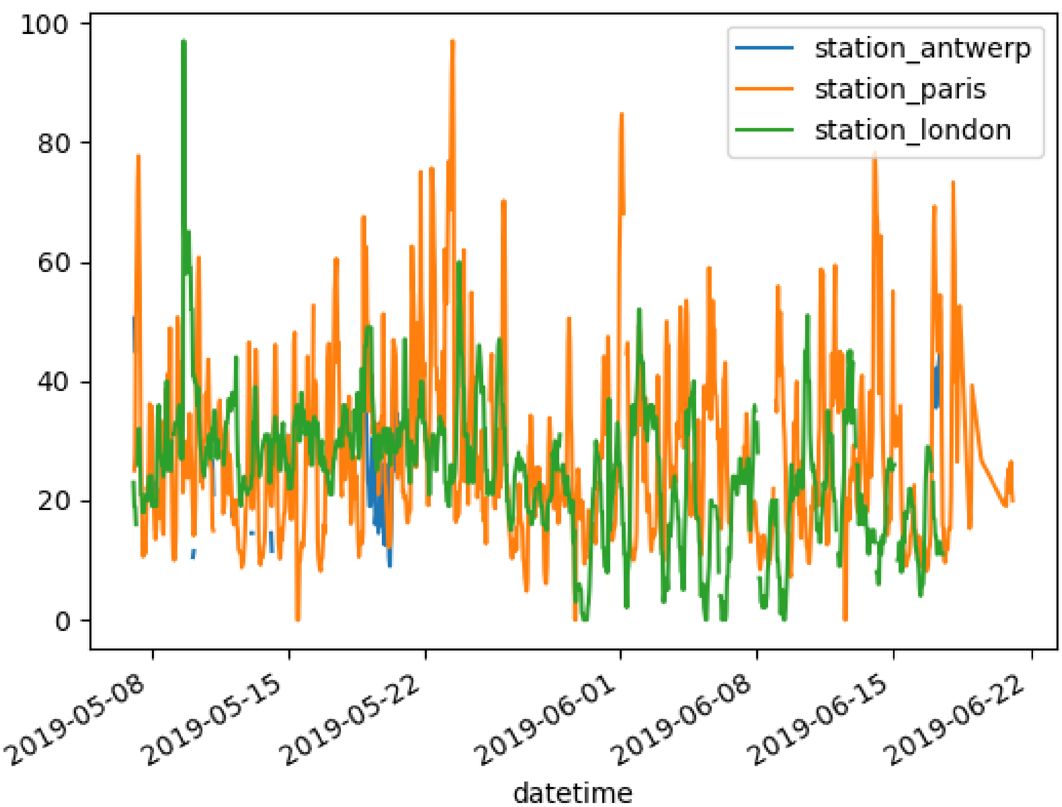
```
In [3]: air_quality = pd.read_csv("data/air_quality_no2.csv", index_col=0, parse_
↪dates=True)

In [4]: air_quality.head()
Out[4]:
                     station_antwerp  station_paris  station_london
datetime
2019-05-07 02:00:00              NaN            NaN            23.0
2019-05-07 03:00:00             50.5           25.0            19.0
2019-05-07 04:00:00             45.0           27.7            19.0
2019-05-07 05:00:00              NaN           50.4            16.0
2019-05-07 06:00:00              NaN           61.9             NaN
```

Note: The usage of the `index_col` and `parse_dates` parameters of the `read_csv` function to define the first (0th) column as index of the resulting `DataFrame` and convert the dates in the column to *`Timestamp`* objects, respectively.

How to create plots in pandas?

I want a quick visual check of the data.

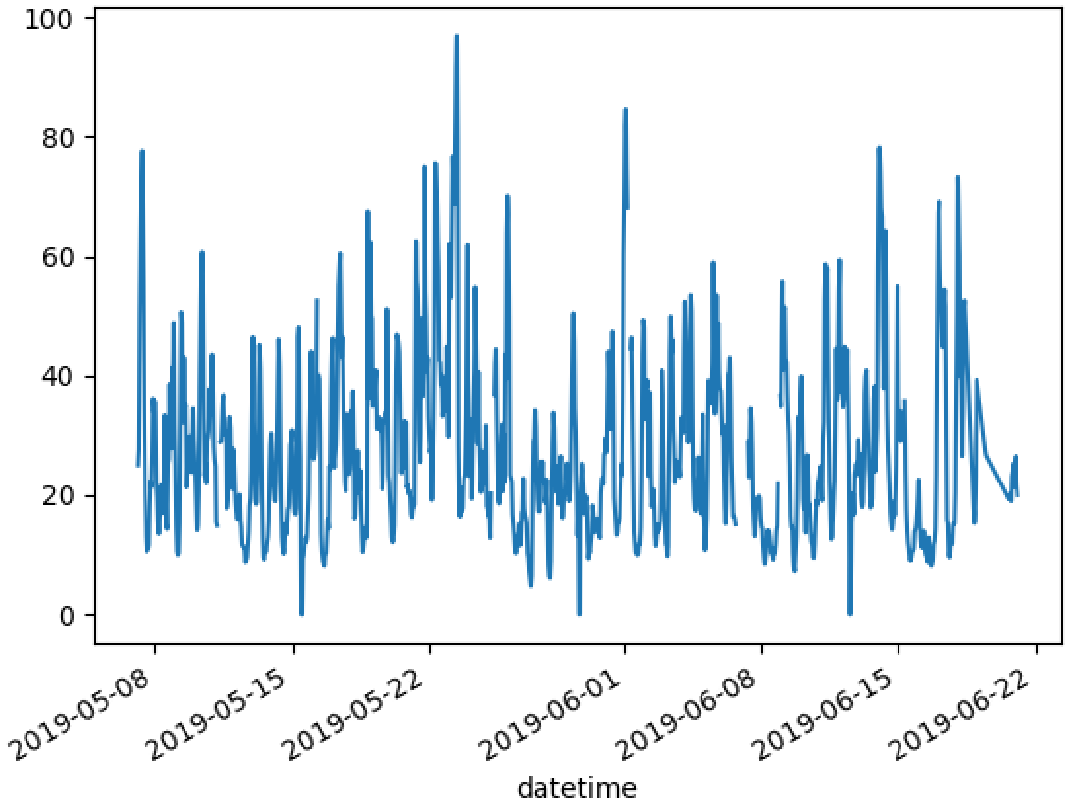
```
In [5]: air_quality.plot()
Out[5]: <AxesSubplot:xlabel='datetime'>
```



With a `DataFrame`, pandas creates by default one line plot for each of the columns with numeric data.

I want to plot only the columns of the data table with the data from Paris.

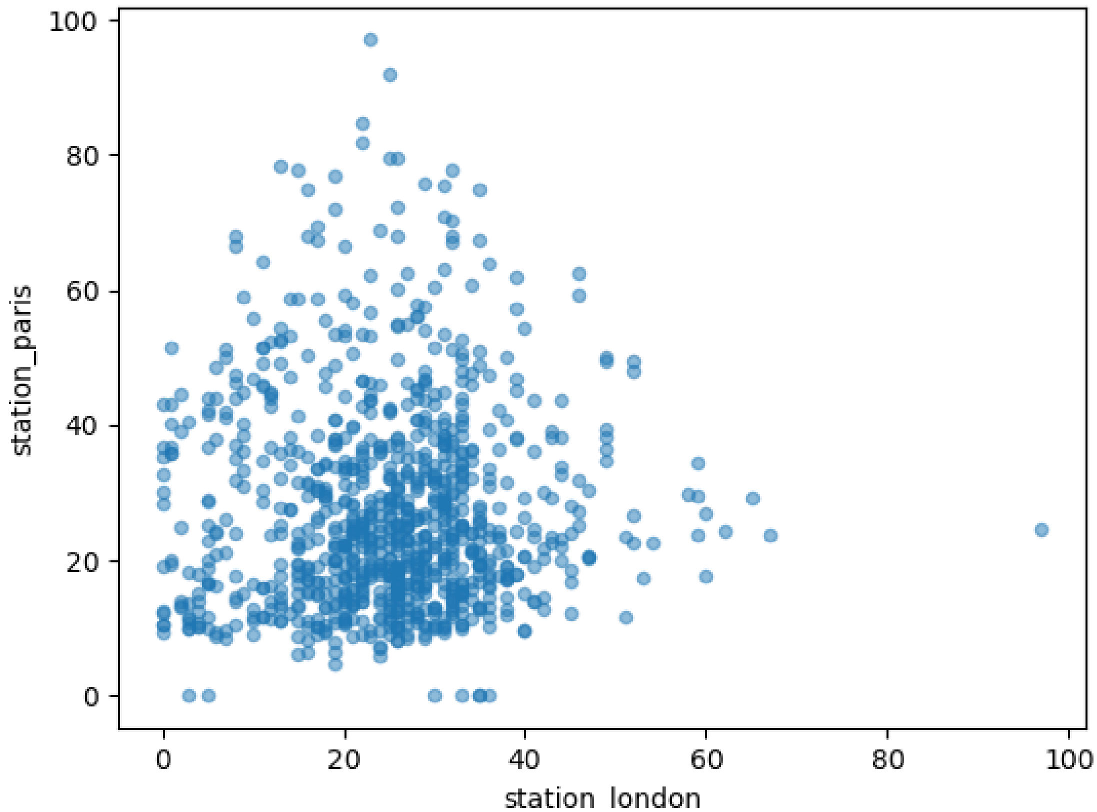
```
In [6]: air_quality["station_paris"].plot()
Out[6]: <AxesSubplot:xlabel='datetime'>
```



To plot a specific column, use the selection method of the *subset data tutorial* in combination with the `plot()` method. Hence, the `plot()` method works on both `Series` and `DataFrame`.

I want to visually compare the $N0_2$ values measured in London versus Paris.

```
In [7]: air_quality.plot.scatter(x="station_london", y="station_paris", alpha=0.5)
Out[7]: <AxesSubplot:xlabel='station_london', ylabel='station_paris'>
```



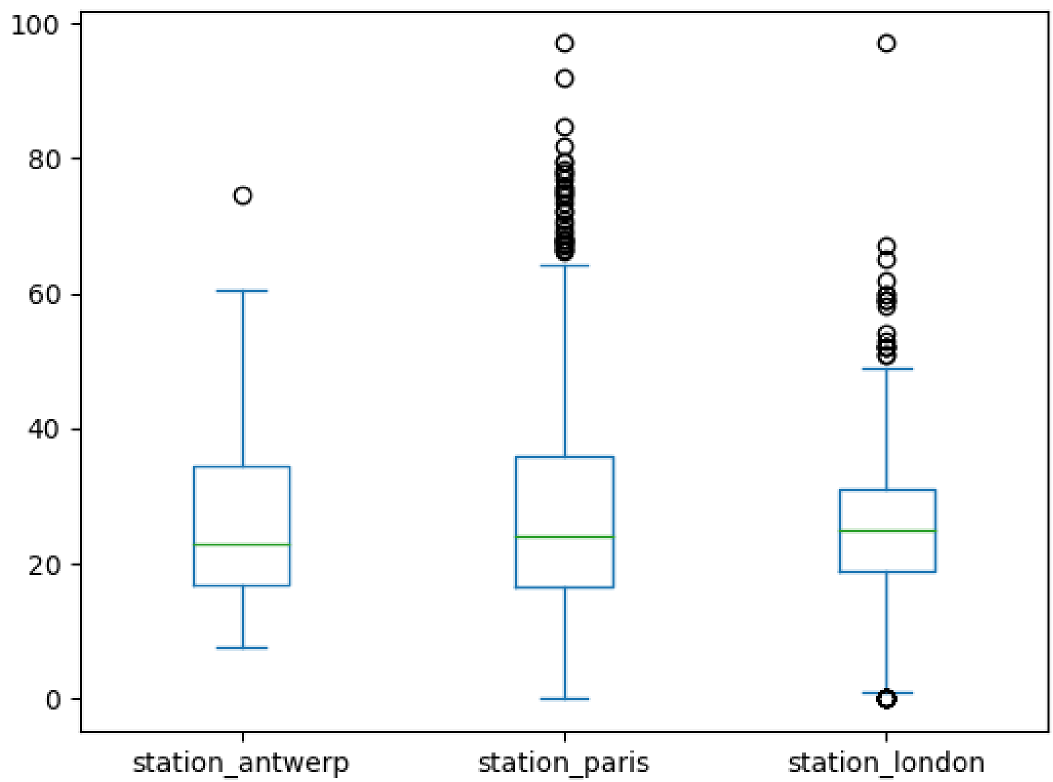
Apart from the default `line` plot when using the `plot` function, a number of alternatives are available to plot data. Let's use some standard Python to get an overview of the available plot methods:

```
In [8]: [
   ...:     method_name
   ...:     for method_name in dir(air_quality.plot)
   ...:     if not method_name.startswith("_")
   ...: ]
   ...:
Out[8]:
['area',
 'bar',
 'barh',
 'box',
 'density',
 'hexbin',
 'hist',
 'kde',
 'line',
 'pie',
 'scatter']
```

Note: In many development environments as well as IPython and Jupyter Notebook, use the TAB button to get an overview of the available methods, for example `air_quality.plot.` + TAB.

One of the options is *DataFrame.plot.box()*, which refers to a boxplot. The `box` method is applicable on the air quality example data:

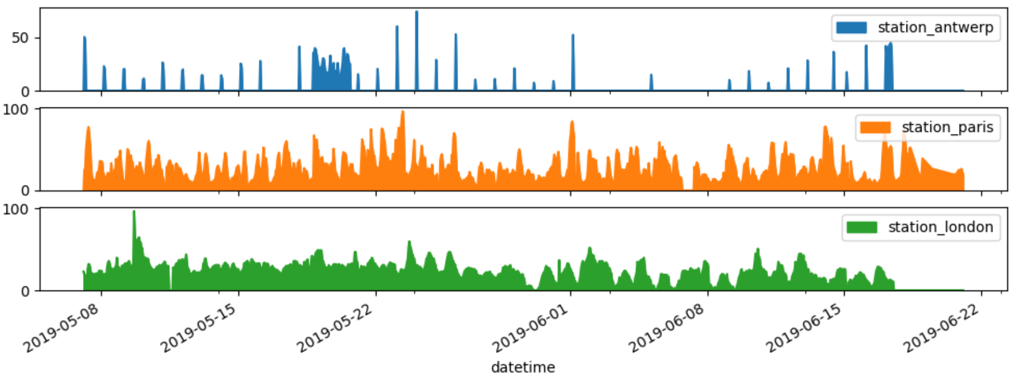
```
In [9]: air_quality.plot.box()
Out[9]: <AxesSubplot:>
```



For an introduction to plots other than the default line plot, see the user guide section about *supported plot styles*.

I want each of the columns in a separate subplot.

```
In [10]: axs = air_quality.plot.area(figsize=(12, 4), subplots=True)
```

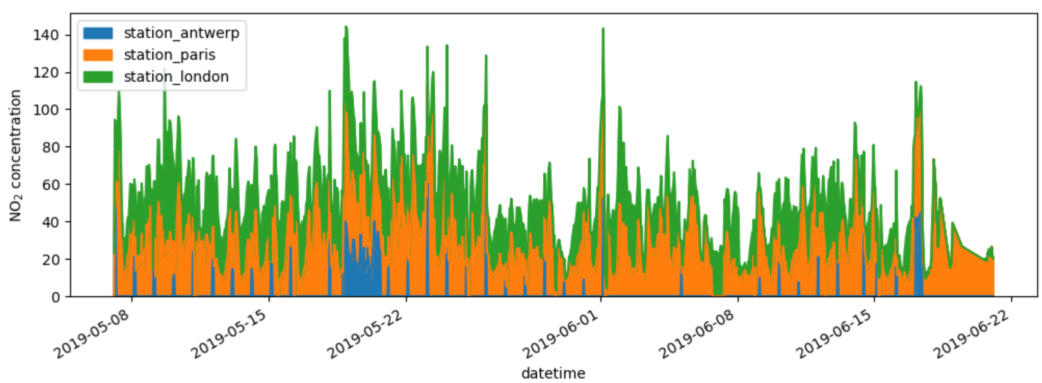


Separate subplots for each of the data columns are supported by the `subplots` argument of the `plot` functions. The builtin options available in each of the pandas plot functions are worth reviewing.

Some more formatting options are explained in the user guide section on *plot formatting*.

I want to further customize, extend or save the resulting plot.

```
In [11]: fig, axs = plt.subplots(figsize=(12, 4))

In [12]: air_quality.plot.area(ax=axs)
Out[12]: <AxesSubplot:xlabel='datetime'>

In [13]: axs.set_ylabel("NO$_2$ concentration")
Out[13]: Text(0, 0.5, 'NO$_2$ concentration')

In [14]: fig.savefig("no2_concentrations.png")
```



Each of the plot objects created by pandas is a matplotlib object. As Matplotlib provides plenty of options to customize plots, making the link between pandas and Matplotlib explicit enables all the power of matplotlib to the plot. This strategy is applied in the previous example:

```
fig, axs = plt.subplots(figsize=(12, 4))        # Create an empty matplotlib Figure and␣
↪Axes
air_quality.plot.area(ax=axs)                   # Use pandas to put the area plot on the␣
↪prepared Figure/Axes
axs.set_ylabel("NO$_2$ concentration")          # Do any matplotlib customization you␣
↪like
```

(continues on next page)

(continued from previous page)

```
fig.savefig("no2_concentrations.png")          # Save the Figure/Axes using the␣
↪existing matplotlib method.
```

- The `.plot.*` methods are applicable on both Series and DataFrames
- By default, each of the columns is plotted as a different element (line, boxplot,…)
- Any plot created by pandas is a Matplotlib object.

A full overview of plotting in pandas is provided in the *visualization pages*.

```
In [1]: import pandas as pd
```

For this tutorial, air quality data about NO_2 is used, made available by openaq and using the py-openaq package. The `air_quality_no2.csv` data set provides NO_2 values for the measurement stations *FR04014*, *BETR801* and *London Westminster* in respectively Paris, Antwerp and London.

```
In [2]: air_quality = pd.read_csv("data/air_quality_no2.csv", index_col=0, parse_
↪dates=True)

In [3]: air_quality.head()
Out[3]: 
                     station_antwerp  station_paris  station_london
datetime                                                           
2019-05-07 02:00:00              NaN            NaN            23.0
2019-05-07 03:00:00             50.5           25.0            19.0
2019-05-07 04:00:00             45.0           27.7            19.0
2019-05-07 05:00:00              NaN           50.4            16.0
2019-05-07 06:00:00              NaN           61.9             NaN
```

How to create new columns derived from existing columns?

I want to express the NO_2 concentration of the station in London in mg/m^3

(*If we assume temperature of 25 degrees Celsius and pressure of 1013 hPa, the conversion factor is 1.882*)

```
In [4]: air_quality["london_mg_per_cubic"] = air_quality["station_london"] * 1.882

In [5]: air_quality.head()
Out[5]: 
                     station_antwerp  station_paris  station_london  london_mg_per_cubic
datetime                                                                                
2019-05-07 02:00:00              NaN            NaN            23.0               43.286
2019-05-07 03:00:00             50.5           25.0            19.0               35.758
2019-05-07 04:00:00             45.0           27.7            19.0               35.758
2019-05-07 05:00:00              NaN           50.4            16.0               30.112
2019-05-07 06:00:00              NaN           61.9             NaN                  NaN
```

To create a new column, use the `[]` brackets with the new column name at the left side of the assignment.

Note: The calculation of the values is done **element_wise**. This means all values in the given column are multiplied by the value 1.882 at once. You do not need to use a loop to iterate each of the rows!

I want to check the ratio of the values in Paris versus Antwerp and save the result in a new column

```
In [6]: air_quality["ratio_paris_antwerp"] = (
   ...:     air_quality["station_paris"] / air_quality["station_antwerp"]
   ...: )
   ...:

In [7]: air_quality.head()
Out[7]:
                     station_antwerp  station_paris  station_london  london_mg_per_cubic␣
↪ ratio_paris_antwerp
datetime                                                                             ␣
↪
2019-05-07 02:00:00              NaN            NaN            23.0               43.286␣
↪                 NaN
2019-05-07 03:00:00             50.5           25.0            19.0               35.758␣
↪            0.495050
2019-05-07 04:00:00             45.0           27.7            19.0               35.758␣
↪            0.615556
2019-05-07 05:00:00              NaN           50.4            16.0               30.112␣
↪                 NaN
2019-05-07 06:00:00              NaN           61.9             NaN                  NaN␣
↪                 NaN
```

The calculation is again element-wise, so the / is applied *for the values in each row*.

Also other mathematical operators (+, -, *, /) or logical operators (<, >, =,...) work element wise. The latter was already used in the *subset data tutorial* to filter rows of a table using a conditional expression.

If you need more advanced logic, you can use arbitrary Python code via `apply()`.

I want to rename the data columns to the corresponding station identifiers used by openAQ

```
In [8]: air_quality_renamed = air_quality.rename(
   ...:     columns={
   ...:         "station_antwerp": "BETR801",
   ...:         "station_paris": "FR04014",
   ...:         "station_london": "London Westminster",
   ...:     }
   ...: )
   ...:
```

```
In [9]: air_quality_renamed.head()
Out[9]:
                     BETR801  FR04014  London Westminster  london_mg_per_cubic  ratio_
↪paris_antwerp
datetime                                                                             ␣
↪
2019-05-07 02:00:00      NaN      NaN                23.0               43.286       ␣
↪       NaN
2019-05-07 03:00:00     50.5     25.0                19.0               35.758       ␣
↪   0.495050
```

(continues on next page)

(continued from previous page)

```
2019-05-07 04:00:00     45.0     27.7                19.0                35.758           ␣
↪  0.615556
2019-05-07 05:00:00      NaN     50.4                16.0                30.112           ␣
↪        NaN
2019-05-07 06:00:00      NaN     61.9                 NaN                   NaN           ␣
↪        NaN
```

The *rename()* function can be used for both row labels and column labels. Provide a dictionary with the keys the current names and the values the new names to update the corresponding names.

The mapping should not be restricted to fixed names only, but can be a mapping function as well. For example, converting the column names to lowercase letters can be done using a function as well:

```
In [10]: air_quality_renamed = air_quality_renamed.rename(columns=str.lower)

In [11]: air_quality_renamed.head()
Out[11]:
                     betr801  fr04014  london westminster  london_mg_per_cubic  ratio_
↪paris_antwerp
datetime                                                                                ␣
↪
2019-05-07 02:00:00      NaN      NaN                23.0                43.286           ␣
↪        NaN
2019-05-07 03:00:00     50.5     25.0                19.0                35.758           ␣
↪  0.495050
2019-05-07 04:00:00     45.0     27.7                19.0                35.758           ␣
↪  0.615556
2019-05-07 05:00:00      NaN     50.4                16.0                30.112           ␣
↪        NaN
2019-05-07 06:00:00      NaN     61.9                 NaN                   NaN           ␣
↪        NaN
```

Details about column or row label renaming is provided in the user guide section on *renaming labels*.

- Create a new column by assigning the output to the DataFrame with a new column name in between the `[]`.
- Operations are element-wise, no need to loop over rows.
- Use `rename` with a dictionary or function to rename row labels or column names.

The user guide contains a separate section on *column addition and deletion*.

```
In [1]: import pandas as pd
```

This tutorial uses the Titanic data set, stored as CSV. The data consists of the following data columns:

- PassengerId: Id of every passenger.
- Survived: This feature have value 0 and 1. 0 for not survived and 1 for survived.
- Pclass: There are 3 classes: Class 1, Class 2 and Class 3.
- Name: Name of passenger.
- Sex: Gender of passenger.
- Age: Age of passenger.
- SibSp: Indication that passenger have siblings and spouse.

- Parch: Whether a passenger is alone or have family.
- Ticket: Ticket number of passenger.
- Fare: Indicating the fare.
- Cabin: The cabin of passenger.
- Embarked: The embarked category.

```
In [2]: titanic = pd.read_csv("data/titanic.csv")

In [3]: titanic.head()
Out[3]:
   PassengerId  Survived  Pclass                                               Name   ␣
↪Sex  ...  Parch            Ticket     Fare Cabin  Embarked
0            1         0       3                            Braund, Mr. Owen Harris   ␣
↪male  ...      0         A/5 21171   7.2500   NaN         S
1            2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th... ␣
↪female  ...      0          PC 17599  71.2833   C85         C
2            3         1       3                             Heikkinen, Miss. Laina ␣
↪female  ...      0  STON/O2. 3101282   7.9250   NaN         S
3            4         1       1       Futrelle, Mrs. Jacques Heath (Lily May Peel) ␣
↪female  ...      0            113803  53.1000  C123         S
4            5         0       3                           Allen, Mr. William Henry   ␣
↪male  ...      0            373450   8.0500   NaN         S

[5 rows x 12 columns]
```

How to calculate summary statistics?

Aggregating statistics

What is the average age of the Titanic passengers?

```
In [4]: titanic["Age"].mean()
Out[4]: 29.69911764705882
```

Different statistics are available and can be applied to columns with numerical data. Operations in general exclude missing data and operate across rows by default.

What is the median age and ticket fare price of the Titanic passengers?

```
In [5]: titanic[["Age", "Fare"]].median()
Out[5]:
Age     28.0000
Fare    14.4542
dtype: float64
```

The statistic applied to multiple columns of a `DataFrame` (the selection of two columns return a `DataFrame`, see the *subset data tutorial*) is calculated for each numeric column.

The aggregating statistic can be calculated for multiple columns at the same time. Remember the `describe` function from *first tutorial*?

```
In [6]: titanic[["Age", "Fare"]].describe()
Out[6]:
              Age        Fare
count  714.000000  891.000000
mean    29.699118   32.204208
std     14.526497   49.693429
min      0.420000    0.000000
25%     20.125000    7.910400
50%     28.000000   14.454200
75%     38.000000   31.000000
max     80.000000  512.329200
```

Instead of the predefined statistics, specific combinations of aggregating statistics for given columns can be defined using the `DataFrame.agg()` method:

```
In [7]: titanic.agg(
   ...:     {
   ...:         "Age": ["min", "max", "median", "skew"],
   ...:         "Fare": ["min", "max", "median", "mean"],
   ...:     }
   ...: )
   ...:
Out[7]:
              Age        Fare
min      0.420000    0.000000
max     80.000000  512.329200
median  28.000000   14.454200
skew     0.389108         NaN
mean          NaN   32.204208
```

Details about descriptive statistics are provided in the user guide section on *descriptive statistics*.

Aggregating statistics grouped by category

What is the average age for male versus female Titanic passengers?

```
In [8]: titanic[["Sex", "Age"]].groupby("Sex").mean()
Out[8]:
              Age
Sex
female  27.915709
male    30.726645
```

As our interest is the average age for each gender, a subselection on these two columns is made first: `titanic[["Sex", "Age"]]`. Next, the `groupby()` method is applied on the `Sex` column to make a group per category. The average age *for each gender* is calculated and returned.

Calculating a given statistic (e.g. `mean` age) *for each category in a column* (e.g. male/female in the `Sex` column) is a common pattern. The `groupby` method is used to support this type of operations. More general, this fits in the more general `split-apply-combine` pattern:

- **Split** the data into groups
- **Apply** a function to each group independently
- **Combine** the results into a data structure

The apply and combine steps are typically done together in pandas.

In the previous example, we explicitly selected the 2 columns first. If not, the `mean` method is applied to each column containing numerical columns:

```
In [9]: titanic.groupby("Sex").mean()
Out[9]:
        PassengerId  Survived    Pclass        Age     SibSp     Parch       Fare
Sex
female   431.028662  0.742038  2.159236  27.915709  0.694268  0.649682  44.479818
male     454.147314  0.188908  2.389948  30.726645  0.429809  0.235702  25.523893
```

It does not make much sense to get the average value of the `Pclass`. if we are only interested in the average age for each gender, the selection of columns (rectangular brackets `[]` as usual) is supported on the grouped data as well:

```
In [10]: titanic.groupby("Sex")["Age"].mean()
Out[10]:
Sex
female    27.915709
male      30.726645
Name: Age, dtype: float64
```

Note: The `Pclass` column contains numerical data but actually represents 3 categories (or factors) with respectively the labels '1', '2' and '3'. Calculating statistics on these does not make much sense. Therefore, pandas provides a `Categorical` data type to handle this type of data. More information is provided in the user guide *Categorical data* section.

What is the mean ticket fare price for each of the sex and cabin class combinations?

```
In [11]: titanic.groupby(["Sex", "Pclass"])["Fare"].mean()
Out[11]:
Sex     Pclass
female  1         106.125798
        2          21.970121
        3          16.118810
male    1          67.226127
        2          19.741782
        3          12.661633
Name: Fare, dtype: float64
```

Grouping can be done by multiple columns at the same time. Provide the column names as a list to the `groupby()` method.

A full description on the split-apply-combine approach is provided in the user guide section on *groupby operations*.

Count number of records by category

What is the number of passengers in each of the cabin classes?

```
In [12]: titanic["Pclass"].value_counts()
Out[12]:
3    491
1    216
2    184
Name: Pclass, dtype: int64
```

The *value_counts()* method counts the number of records for each category in a column.

The function is a shortcut, as it is actually a groupby operation in combination with counting of the number of records within each group:

```
In [13]: titanic.groupby("Pclass")["Pclass"].count()
Out[13]:
Pclass
1    216
2    184
3    491
Name: Pclass, dtype: int64
```

Note: Both `size` and `count` can be used in combination with `groupby`. Whereas `size` includes `NaN` values and just provides the number of rows (size of the table), `count` excludes the missing values. In the `value_counts` method, use the `dropna` argument to include or exclude the `NaN` values.

The user guide has a dedicated section on `value_counts` , see page on *discretization*.

- Aggregation statistics can be calculated on entire columns or rows
- `groupby` provides the power of the *split-apply-combine* pattern
- `value_counts` is a convenient shortcut to count the number of entries in each category of a variable

A full description on the split-apply-combine approach is provided in the user guide pages about *groupby operations*.

```
In [1]: import pandas as pd
```

This tutorial uses the Titanic data set, stored as CSV. The data consists of the following data columns:

- PassengerId: Id of every passenger.
- Survived: This feature have value 0 and 1. 0 for not survived and 1 for survived.
- Pclass: There are 3 classes: Class 1, Class 2 and Class 3.
- Name: Name of passenger.
- Sex: Gender of passenger.
- Age: Age of passenger.
- SibSp: Indication that passenger have siblings and spouse.
- Parch: Whether a passenger is alone or have family.

- Ticket: Ticket number of passenger.
- Fare: Indicating the fare.
- Cabin: The cabin of passenger.
- Embarked: The embarked category.

```
In [2]: titanic = pd.read_csv("data/titanic.csv")

In [3]: titanic.head()
Out[3]:
   PassengerId  Survived  Pclass                                               Name  ␣
↪Sex  ...  Parch            Ticket     Fare Cabin  Embarked
0            1         0       3                            Braund, Mr. Owen Harris  ␣
↪male  ...      0         A/5 21171   7.2500   NaN         S
1            2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th... ␣
↪female  ...      0          PC 17599  71.2833   C85         C
2            3         1       3                             Heikkinen, Miss. Laina ␣
↪female  ...      0  STON/O2. 3101282   7.9250   NaN         S
3            4         1       1       Futrelle, Mrs. Jacques Heath (Lily May Peel) ␣
↪female  ...      0            113803  53.1000  C123         S
4            5         0       3                           Allen, Mr. William Henry  ␣
↪male  ...      0            373450   8.0500   NaN         S

[5 rows x 12 columns]
```

This tutorial uses air quality data about NO_2 and Particulate matter less than 2.5 micrometers, made available by openaq and using the py-openaq package. The `air_quality_long.csv` data set provides NO_2 and PM_{25} values for the measurement stations *FR04014*, *BETR801* and *London Westminster* in respectively Paris, Antwerp and London.

The air-quality data set has the following columns:

- city: city where the sensor is used, either Paris, Antwerp or London
- country: country where the sensor is used, either FR, BE or GB
- location: the id of the sensor, either *FR04014*, *BETR801* or *London Westminster*
- parameter: the parameter measured by the sensor, either NO_2 or Particulate matter
- value: the measured value
- unit: the unit of the measured parameter, in this case 'µg/m^3'

and the index of the `DataFrame` is `datetime`, the datetime of the measurement.

Note: The air-quality data is provided in a so-called *long format* data representation with each observation on a separate row and each variable a separate column of the data table. The long/narrow format is also known as the tidy data format.

```
In [4]: air_quality = pd.read_csv(
   ...:     "data/air_quality_long.csv", index_col="date.utc", parse_dates=True
   ...: )
   ...:

In [5]: air_quality.head()
```

(continues on next page)

(continued from previous page)

```
Out[5]:
                                city country location parameter  value   unit
date.utc
2019-06-18 06:00:00+00:00  Antwerpen      BE  BETR801      pm25   18.0  µg/m³
2019-06-17 08:00:00+00:00  Antwerpen      BE  BETR801      pm25    6.5  µg/m³
2019-06-17 07:00:00+00:00  Antwerpen      BE  BETR801      pm25   18.5  µg/m³
2019-06-17 06:00:00+00:00  Antwerpen      BE  BETR801      pm25   16.0  µg/m³
2019-06-17 05:00:00+00:00  Antwerpen      BE  BETR801      pm25    7.5  µg/m³
```

How to reshape the layout of tables?

Sort table rows

I want to sort the Titanic data according to the age of the passengers.

```
In [6]: titanic.sort_values(by="Age").head()
Out[6]:
     PassengerId  Survived  Pclass                             Name     Sex   Age  SibSp␣
↪ Parch  Ticket     Fare Cabin Embarked
803          804         1       3  Thomas, Master. Assad Alexander    male  0.42      0␣
↪     1    2625   8.5167   NaN        C
755          756         1       2        Hamalainen, Master. Viljo    male  0.67      1␣
↪     1  250649  14.5000   NaN        S
644          645         1       3           Baclini, Miss. Eugenie  female  0.75      2␣
↪     1    2666  19.2583   NaN        C
469          470         1       3    Baclini, Miss. Helene Barbara  female  0.75      2␣
↪     1    2666  19.2583   NaN        C
78            79         1       2    Caldwell, Master. Alden Gates    male  0.83      0␣
↪     2  248738  29.0000   NaN        S
```

I want to sort the Titanic data according to the cabin class and age in descending order.

```
In [7]: titanic.sort_values(by=['Pclass', 'Age'], ascending=False).head()
Out[7]:
     PassengerId  Survived  Pclass                        Name     Sex   Age  SibSp ␣
↪Parch  Ticket    Fare Cabin Embarked
851          852         0       3         Svensson, Mr. Johan    male  74.0      0       ␣
↪0  347060  7.7750   NaN        S
116          117         0       3        Connors, Mr. Patrick    male  70.5      0       ␣
↪0  370369  7.7500   NaN        Q
280          281         0       3            Duane, Mr. Frank    male  65.0      0       ␣
↪0  336439  7.7500   NaN        Q
483          484         1       3      Turkula, Mrs. (Hedwig)  female  63.0      0       ␣
↪0    4134  9.5875   NaN        S
326          327         0       3  Nysveen, Mr. Johan Hansen    male  61.0      0       ␣
↪0  345364  6.2375   NaN        S
```

With *Series.sort_values()*, the rows in the table are sorted according to the defined column(s). The index will follow the row order.

More details about sorting of tables is provided in the using guide section on *sorting data*.

Long to wide table format

Let's use a small subset of the air quality data set. We focus on NO_2 data and only use the first two measurements of each location (i.e. the head of each group). The subset of data will be called `no2_subset`

```
# filter for no2 data only
In [8]: no2 = air_quality[air_quality["parameter"] == "no2"]
```

```
# use 2 measurements (head) for each location (groupby)
In [9]: no2_subset = no2.sort_index().groupby(["location"]).head(2)

In [10]: no2_subset
Out[10]:
                                city country            location parameter  value   unit
date.utc
2019-04-09 01:00:00+00:00  Antwerpen      BE             BETR801       no2   22.5  µg/m³
2019-04-09 01:00:00+00:00      Paris      FR             FR04014       no2   24.4  µg/m³
2019-04-09 02:00:00+00:00     London      GB  London Westminster       no2   67.0  µg/m³
2019-04-09 02:00:00+00:00  Antwerpen      BE             BETR801       no2   53.5  µg/m³
2019-04-09 02:00:00+00:00      Paris      FR             FR04014       no2   27.4  µg/m³
2019-04-09 03:00:00+00:00     London      GB  London Westminster       no2   67.0  µg/m³
```

I want the values for the three stations as separate columns next to each other

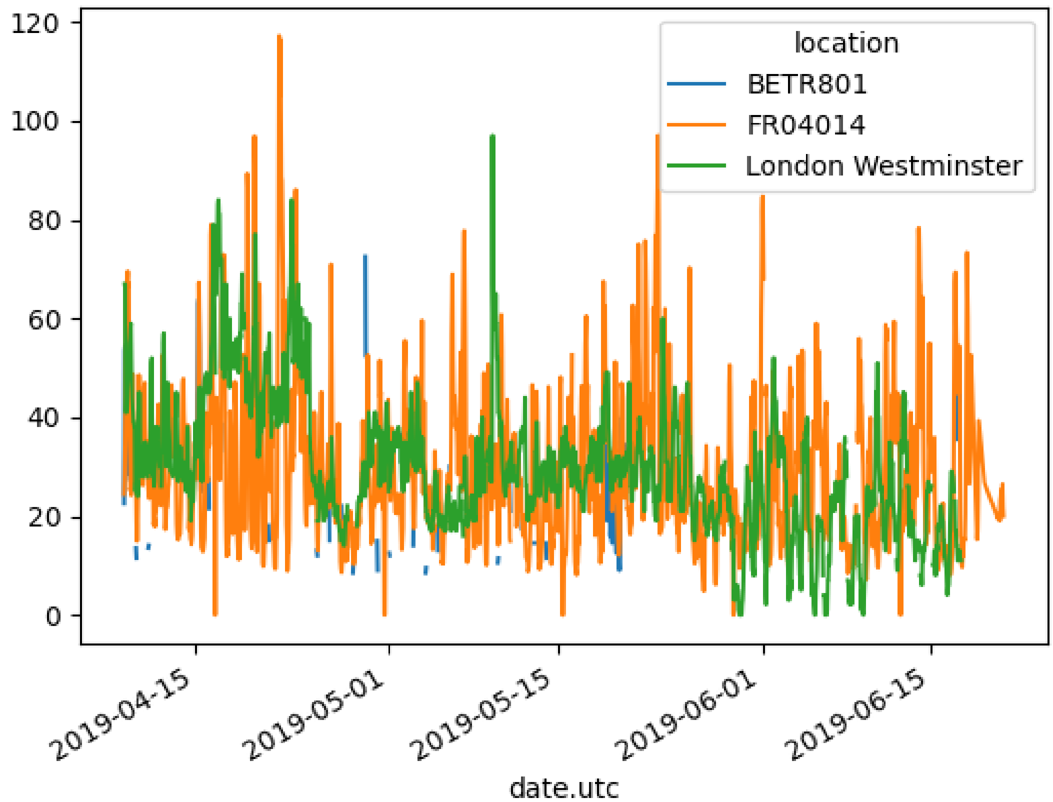
```
In [11]: no2_subset.pivot(columns="location", values="value")
Out[11]:
location                   BETR801  FR04014  London Westminster
date.utc
2019-04-09 01:00:00+00:00     22.5     24.4                 NaN
2019-04-09 02:00:00+00:00     53.5     27.4                67.0
2019-04-09 03:00:00+00:00      NaN      NaN                67.0
```

The `pivot()` function is purely reshaping of the data: a single value for each index/column combination is required.

As pandas support plotting of multiple columns (see *plotting tutorial*) out of the box, the conversion from *long* to *wide* table format enables the plotting of the different time series at the same time:

```
In [12]: no2.head()
Out[12]:
                            city country location parameter  value   unit
date.utc
2019-06-21 00:00:00+00:00  Paris      FR  FR04014       no2   20.0  µg/m³
2019-06-20 23:00:00+00:00  Paris      FR  FR04014       no2   21.8  µg/m³
2019-06-20 22:00:00+00:00  Paris      FR  FR04014       no2   26.5  µg/m³
2019-06-20 21:00:00+00:00  Paris      FR  FR04014       no2   24.9  µg/m³
2019-06-20 20:00:00+00:00  Paris      FR  FR04014       no2   21.4  µg/m³
```

```
In [13]: no2.pivot(columns="location", values="value").plot()
Out[13]: <AxesSubplot:xlabel='date.utc'>
```



Note: When the `index` parameter is not defined, the existing index (row labels) is used.

For more information about *pivot()*, see the user guide section on *pivoting DataFrame objects*.

Pivot table

I want the mean concentrations for NO_2 and $PM_{2.5}$ in each of the stations in table form

```
In [14]: air_quality.pivot_table(
   ....:     values="value", index="location", columns="parameter", aggfunc="mean"
   ....: )
   ....:
Out[14]:
parameter                 no2       pm25
location
BETR801             26.950920  23.169492
FR04014             29.374284        NaN
London Westminster  29.740050  13.443568
```

In the case of *pivot()*, the data is only rearranged. When multiple values need to be aggregated (in this specific case, the values on different time steps) *pivot_table()* can be used, providing an aggregation function (e.g. mean) on how to combine these values.

Pivot table is a well known concept in spreadsheet software. When interested in summary columns for each variable separately as well, put the `margin` parameter to `True`:

```
In [15]: air_quality.pivot_table(
   ....:     values="value",
   ....:     index="location",
   ....:     columns="parameter",
   ....:     aggfunc="mean",
   ....:     margins=True,
   ....: )
   ....:
Out[15]:
parameter                 no2       pm25        All
location
BETR801             26.950920  23.169492  24.982353
FR04014             29.374284        NaN  29.374284
London Westminster  29.740050  13.443568  21.491708
All                 29.430316  14.386849  24.222743
```

For more information about *pivot_table()*, see the user guide section on *pivot tables*.

Note: In case you are wondering, *pivot_table()* is indeed directly linked to *groupby()*. The same result can be derived by grouping on both `parameter` and `location`:

```
air_quality.groupby(["parameter", "location"]).mean()
```

Have a look at *groupby()* in combination with *unstack()* at the user guide section on *combining stats and groupby*.

Wide to long format

Starting again from the wide format table created in the previous section:

```
In [16]: no2_pivoted = no2.pivot(columns="location", values="value").reset_index()

In [17]: no2_pivoted.head()
Out[17]:
location                 date.utc  BETR801  FR04014  London Westminster
0        2019-04-09 01:00:00+00:00     22.5     24.4                 NaN
1        2019-04-09 02:00:00+00:00     53.5     27.4                67.0
2        2019-04-09 03:00:00+00:00     54.5     34.2                67.0
3        2019-04-09 04:00:00+00:00     34.5     48.5                41.0
4        2019-04-09 05:00:00+00:00     46.5     59.5                41.0
```

I want to collect all air quality NO_2 measurements in a single column (long format)

```
In [18]: no_2 = no2_pivoted.melt(id_vars="date.utc")

```

(continues on next page)

(continued from previous page)

```
In [19]: no_2.head()
Out[19]: 
                   date.utc location  value
0 2019-04-09 01:00:00+00:00  BETR801   22.5
1 2019-04-09 02:00:00+00:00  BETR801   53.5
2 2019-04-09 03:00:00+00:00  BETR801   54.5
3 2019-04-09 04:00:00+00:00  BETR801   34.5
4 2019-04-09 05:00:00+00:00  BETR801   46.5
```

The *pandas.melt()* method on a DataFrame converts the data table from wide format to long format. The column headers become the variable names in a newly created column.

The solution is the short version on how to apply *pandas.melt()*. The method will *melt* all columns NOT mentioned in id_vars together into two columns: A column with the column header names and a column with the values itself. The latter column gets by default the name value.

The *pandas.melt()* method can be defined in more detail:

```
In [20]: no_2 = no2_pivoted.melt(
   ....:     id_vars="date.utc",
   ....:     value_vars=["BETR801", "FR04014", "London Westminster"],
   ....:     value_name="NO_2",
   ....:     var_name="id_location",
   ....: )
   ....: 

In [21]: no_2.head()
Out[21]: 
                   date.utc id_location  NO_2
0 2019-04-09 01:00:00+00:00     BETR801  22.5
1 2019-04-09 02:00:00+00:00     BETR801  53.5
2 2019-04-09 03:00:00+00:00     BETR801  54.5
3 2019-04-09 04:00:00+00:00     BETR801  34.5
4 2019-04-09 05:00:00+00:00     BETR801  46.5
```

The result in the same, but in more detail defined:

- value_vars defines explicitly which columns to *melt* together
- value_name provides a custom column name for the values column instead of the default column name value
- var_name provides a custom column name for the column collecting the column header names. Otherwise it takes the index name or a default variable

Hence, the arguments value_name and var_name are just user-defined names for the two generated columns. The columns to melt are defined by id_vars and value_vars.

Conversion from wide to long format with *pandas.melt()* is explained in the user guide section on *reshaping by melt*.

- Sorting by one or more columns is supported by sort_values
- The pivot function is purely restructuring of the data, pivot_table supports aggregations
- The reverse of pivot (long to wide format) is melt (wide to long format)

A full overview is available in the user guide on the pages about *reshaping and pivoting*.

```
In [1]: import pandas as pd
```

For this tutorial, air quality data about NO_2 is used, made available by openaq and downloaded using the py-openaq package.

The `air_quality_no2_long.csv` data set provides NO_2 values for the measurement stations *FR04014*, *BETR801* and *London Westminster* in respectively Paris, Antwerp and London.

```
In [2]: air_quality_no2 = pd.read_csv("data/air_quality_no2_long.csv",
   ...:                               parse_dates=True)
   ...:

In [3]: air_quality_no2 = air_quality_no2[["date.utc", "location",
   ...:                                    "parameter", "value"]]
   ...:

In [4]: air_quality_no2.head()
Out[4]:
                    date.utc location parameter  value
0  2019-06-21 00:00:00+00:00  FR04014       no2   20.0
1  2019-06-20 23:00:00+00:00  FR04014       no2   21.8
2  2019-06-20 22:00:00+00:00  FR04014       no2   26.5
3  2019-06-20 21:00:00+00:00  FR04014       no2   24.9
4  2019-06-20 20:00:00+00:00  FR04014       no2   21.4
```

For this tutorial, air quality data about Particulate matter less than 2.5 micrometers is used, made available by openaq and downloaded using the py-openaq package.

The `air_quality_pm25_long.csv` data set provides PM_{25} values for the measurement stations *FR04014*, *BETR801* and *London Westminster* in respectively Paris, Antwerp and London.

```
In [5]: air_quality_pm25 = pd.read_csv("data/air_quality_pm25_long.csv",
   ...:                                parse_dates=True)
   ...:

In [6]: air_quality_pm25 = air_quality_pm25[["date.utc", "location",
   ...:                                      "parameter", "value"]]
   ...:

In [7]: air_quality_pm25.head()
Out[7]:
                    date.utc location parameter  value
0  2019-06-18 06:00:00+00:00  BETR801      pm25   18.0
1  2019-06-17 08:00:00+00:00  BETR801      pm25    6.5
2  2019-06-17 07:00:00+00:00  BETR801      pm25   18.5
3  2019-06-17 06:00:00+00:00  BETR801      pm25   16.0
4  2019-06-17 05:00:00+00:00  BETR801      pm25    7.5
```

How to combine data from multiple tables?

Concatenating objects

I want to combine the measurements of NO_2 and PM_{25}, two tables with a similar structure, in a single table

```
In [8]: air_quality = pd.concat([air_quality_pm25, air_quality_no2], axis=0)

In [9]: air_quality.head()
Out[9]:
                    date.utc location parameter  value
0  2019-06-18 06:00:00+00:00  BETR801      pm25   18.0
1  2019-06-17 08:00:00+00:00  BETR801      pm25    6.5
2  2019-06-17 07:00:00+00:00  BETR801      pm25   18.5
3  2019-06-17 06:00:00+00:00  BETR801      pm25   16.0
4  2019-06-17 05:00:00+00:00  BETR801      pm25    7.5
```

The *concat()* function performs concatenation operations of multiple tables along one of the axis (row-wise or column-wise).

By default concatenation is along axis 0, so the resulting table combines the rows of the input tables. Let's check the shape of the original and the concatenated tables to verify the operation:

```
In [10]: print('Shape of the ``air_quality_pm25`` table: ', air_quality_pm25.shape)
Shape of the ``air_quality_pm25`` table:  (1110, 4)

In [11]: print('Shape of the ``air_quality_no2`` table: ', air_quality_no2.shape)
Shape of the ``air_quality_no2`` table:  (2068, 4)

In [12]: print('Shape of the resulting ``air_quality`` table: ', air_quality.shape)
Shape of the resulting ``air_quality`` table:  (3178, 4)
```

Hence, the resulting table has 3178 = 1110 + 2068 rows.

Note: The **axis** argument will return in a number of pandas methods that can be applied **along an axis**. A `DataFrame` has two corresponding axes: the first running vertically downwards across rows (axis 0), and the second running horizontally across columns (axis 1). Most operations like concatenation or summary statistics are by default across rows (axis 0), but can be applied across columns as well.

Sorting the table on the datetime information illustrates also the combination of both tables, with the `parameter` column defining the origin of the table (either `no2` from table `air_quality_no2` or `pm25` from table `air_quality_pm25`):

```
In [13]: air_quality = air_quality.sort_values("date.utc")

In [14]: air_quality.head()
Out[14]:
                       date.utc            location parameter  value
2067  2019-05-07 01:00:00+00:00  London Westminster       no2   23.0
1003  2019-05-07 01:00:00+00:00             FR04014       no2   25.0
100   2019-05-07 01:00:00+00:00             BETR801      pm25   12.5
```

(continues on next page)

(continued from previous page)

```
1098  2019-05-07 01:00:00+00:00             BETR801       no2    50.5
1109  2019-05-07 01:00:00+00:00  London Westminster      pm25     8.0
```

In this specific example, the `parameter` column provided by the data ensures that each of the original tables can be identified. This is not always the case. the `concat` function provides a convenient solution with the `keys` argument, adding an additional (hierarchical) row index. For example:

```
In [15]: air_quality_ = pd.concat([air_quality_pm25, air_quality_no2], keys=["PM25", "NO2
↪"])
```

```
In [16]: air_quality_.head()
Out[16]: 
                         date.utc location parameter  value
PM25 0  2019-06-18 06:00:00+00:00  BETR801      pm25   18.0
     1  2019-06-17 08:00:00+00:00  BETR801      pm25    6.5
     2  2019-06-17 07:00:00+00:00  BETR801      pm25   18.5
     3  2019-06-17 06:00:00+00:00  BETR801      pm25   16.0
     4  2019-06-17 05:00:00+00:00  BETR801      pm25    7.5
```

Note: The existence of multiple row/column indices at the same time has not been mentioned within these tutorials. *Hierarchical indexing* or *MultiIndex* is an advanced and powerful pandas feature to analyze higher dimensional data.

Multi-indexing is out of scope for this pandas introduction. For the moment, remember that the function `reset_index` can be used to convert any level of an index to a column, e.g. `air_quality.reset_index(level=0)`

Feel free to dive into the world of multi-indexing at the user guide section on *advanced indexing*.

More options on table concatenation (row and column wise) and how `concat` can be used to define the logic (union or intersection) of the indexes on the other axes is provided at the section on *object concatenation*.

Join tables using a common identifier

Add the station coordinates, provided by the stations metadata table, to the corresponding rows in the measurements table.

Warning: The air quality measurement station coordinates are stored in a data file `air_quality_stations.csv`, downloaded using the py-openaq package.

```
In [17]: stations_coord = pd.read_csv("data/air_quality_stations.csv")

In [18]: stations_coord.head()
Out[18]: 
  location  coordinates.latitude  coordinates.longitude
0  BELAL01              51.23619                4.38522
1  BELHB23              51.17030                4.34100
2  BELLD01              51.10998                5.00486
3  BELLD02              51.12038                5.02155
4  BELR833              51.32766                4.36226
```

Note: The stations used in this example (FR04014, BETR801 and London Westminster) are just three entries enlisted in the metadata table. We only want to add the coordinates of these three to the measurements table, each on the corresponding rows of the `air_quality` table.

```
In [19]: air_quality.head()
Out[19]:
                       date.utc             location parameter  value
2067  2019-05-07 01:00:00+00:00  London Westminster       no2   23.0
1003  2019-05-07 01:00:00+00:00             FR04014       no2   25.0
100   2019-05-07 01:00:00+00:00             BETR801      pm25   12.5
1098  2019-05-07 01:00:00+00:00             BETR801       no2   50.5
1109  2019-05-07 01:00:00+00:00  London Westminster      pm25    8.0
```

```
In [20]: air_quality = pd.merge(air_quality, stations_coord, how="left", on="location")

In [21]: air_quality.head()
Out[21]:
                    date.utc            location parameter  value  coordinates.latitude ␣
↪coordinates.longitude
0  2019-05-07 01:00:00+00:00  London Westminster       no2   23.0              51.49467 ␣
↪             -0.13193
1  2019-05-07 01:00:00+00:00             FR04014       no2   25.0              48.83724 ␣
↪              2.39390
2  2019-05-07 01:00:00+00:00             FR04014       no2   25.0              48.83722 ␣
↪              2.39390
3  2019-05-07 01:00:00+00:00             BETR801      pm25   12.5              51.20966 ␣
↪              4.43182
4  2019-05-07 01:00:00+00:00             BETR801       no2   50.5              51.20966 ␣
↪              4.43182
```

Using the *merge()* function, for each of the rows in the `air_quality` table, the corresponding coordinates are added from the `air_quality_stations_coord` table. Both tables have the column `location` in common which is used as a key to combine the information. By choosing the `left` join, only the locations available in the `air_quality` (left) table, i.e. FR04014, BETR801 and London Westminster, end up in the resulting table. The `merge` function supports multiple join options similar to database-style operations.

Add the parameter full description and name, provided by the parameters metadata table, to the measurements table

Warning: The air quality parameters metadata are stored in a data file `air_quality_parameters.csv`, downloaded using the py-openaq package.

```
In [22]: air_quality_parameters = pd.read_csv("data/air_quality_parameters.csv")

In [23]: air_quality_parameters.head()
Out[23]:
     id                                description  name
0    bc                               Black Carbon    BC
1    co                            Carbon Monoxide    CO
2   no2                           Nitrogen Dioxide   NO2
3    o3                                      Ozone    O3
```

(continues on next page)

(continued from previous page)

```
4  pm10  Particulate matter less than 10 micrometers in...  PM10
```

```
In [24]: air_quality = pd.merge(air_quality, air_quality_parameters,
   ....:                        how='left', left_on='parameter', right_on='id')
   ....:

In [25]: air_quality.head()
Out[25]:
                    date.utc            location parameter  ...    id                    ␣
↪                   description   name
0  2019-05-07 01:00:00+00:00  London Westminster       no2  ...   no2                    ␣
↪                Nitrogen Dioxide    NO2
1  2019-05-07 01:00:00+00:00             FR04014       no2  ...   no2                    ␣
↪                Nitrogen Dioxide    NO2
2  2019-05-07 01:00:00+00:00             FR04014       no2  ...   no2                    ␣
↪                Nitrogen Dioxide    NO2
3  2019-05-07 01:00:00+00:00             BETR801      pm25  ...  pm25  Particulate␣
↪matter less than 2.5 micrometers i...   PM2.5
4  2019-05-07 01:00:00+00:00             BETR801       no2  ...   no2                    ␣
↪                Nitrogen Dioxide    NO2

[5 rows x 9 columns]
```

Compared to the previous example, there is no common column name. However, the `parameter` column in the `air_quality` table and the `id` column in the `air_quality_parameters_name` both provide the measured variable in a common format. The `left_on` and `right_on` arguments are used here (instead of just `on`) to make the link between the two tables.

pandas supports also inner, outer, and right joins. More information on join/merge of tables is provided in the user guide section on *database style merging of tables*. Or have a look at the *comparison with SQL* page.

- Multiple tables can be concatenated both column-wise and row-wise using the `concat` function.
- For database-like merging/joining of tables, use the `merge` function.

See the user guide for a full description of the various *facilities to combine data tables*.

```
In [1]: import pandas as pd

In [2]: import matplotlib.pyplot as plt
```

For this tutorial, air quality data about NO_2 and Particulate matter less than 2.5 micrometers is used, made available by openaq and downloaded using the py-openaq package. The `air_quality_no2_long.csv"` data set provides NO_2 values for the measurement stations *FR04014*, *BETR801* and *London Westminster* in respectively Paris, Antwerp and London.

```
In [3]: air_quality = pd.read_csv("data/air_quality_no2_long.csv")

In [4]: air_quality = air_quality.rename(columns={"date.utc": "datetime"})

In [5]: air_quality.head()
Out[5]:
    city country                   datetime location parameter  value   unit
0  Paris      FR  2019-06-21 00:00:00+00:00  FR04014       no2   20.0  µg/m³
```

(continues on next page)

(continued from previous page)

```
1  Paris      FR  2019-06-20 23:00:00+00:00  FR04014       no2   21.8  µg/m³
2  Paris      FR  2019-06-20 22:00:00+00:00  FR04014       no2   26.5  µg/m³
3  Paris      FR  2019-06-20 21:00:00+00:00  FR04014       no2   24.9  µg/m³
4  Paris      FR  2019-06-20 20:00:00+00:00  FR04014       no2   21.4  µg/m³
```

```
In [6]: air_quality.city.unique()
Out[6]: array(['Paris', 'Antwerpen', 'London'], dtype=object)
```

How to handle time series data with ease?

Using pandas datetime properties

I want to work with the dates in the column `datetime` as datetime objects instead of plain text

```
In [7]: air_quality["datetime"] = pd.to_datetime(air_quality["datetime"])

In [8]: air_quality["datetime"]
Out[8]:
0       2019-06-21 00:00:00+00:00
1       2019-06-20 23:00:00+00:00
2       2019-06-20 22:00:00+00:00
3       2019-06-20 21:00:00+00:00
4       2019-06-20 20:00:00+00:00
                   ...
2063    2019-05-07 06:00:00+00:00
2064    2019-05-07 04:00:00+00:00
2065    2019-05-07 03:00:00+00:00
2066    2019-05-07 02:00:00+00:00
2067    2019-05-07 01:00:00+00:00
Name: datetime, Length: 2068, dtype: datetime64[ns, UTC]
```

Initially, the values in `datetime` are character strings and do not provide any datetime operations (e.g. extract the year, day of the week,…). By applying the `to_datetime` function, pandas interprets the strings and convert these to datetime (i.e. `datetime64[ns, UTC]`) objects. In pandas we call these datetime objects similar to `datetime.datetime` from the standard library as *pandas.Timestamp*.

Note: As many data sets do contain datetime information in one of the columns, pandas input function like *pandas.read_csv()* and *pandas.read_json()* can do the transformation to dates when reading the data using the `parse_dates` parameter with a list of the columns to read as Timestamp:

```
pd.read_csv("../data/air_quality_no2_long.csv", parse_dates=["datetime"])
```

Why are these *pandas.Timestamp* objects useful? Let's illustrate the added value with some example cases.

What is the start and end date of the time series data set we are working with?

```
In [9]: air_quality["datetime"].min(), air_quality["datetime"].max()
Out[9]:
(Timestamp('2019-05-07 01:00:00+0000', tz='UTC'),
 Timestamp('2019-06-21 00:00:00+0000', tz='UTC'))
```

Using *pandas.Timestamp* for datetimes enables us to calculate with date information and make them comparable. Hence, we can use this to get the length of our time series:

```
In [10]: air_quality["datetime"].max() - air_quality["datetime"].min()
Out[10]: Timedelta('44 days 23:00:00')
```

The result is a *pandas.Timedelta* object, similar to `datetime.timedelta` from the standard Python library and defining a time duration.

The various time concepts supported by pandas are explained in the user guide section on *time related concepts*.

I want to add a new column to the `DataFrame` containing only the month of the measurement

```
In [11]: air_quality["month"] = air_quality["datetime"].dt.month

In [12]: air_quality.head()
Out[12]:
    city country                  datetime location parameter  value   unit  month
0  Paris      FR 2019-06-21 00:00:00+00:00  FR04014       no2   20.0  µg/m³      6
1  Paris      FR 2019-06-20 23:00:00+00:00  FR04014       no2   21.8  µg/m³      6
2  Paris      FR 2019-06-20 22:00:00+00:00  FR04014       no2   26.5  µg/m³      6
3  Paris      FR 2019-06-20 21:00:00+00:00  FR04014       no2   24.9  µg/m³      6
4  Paris      FR 2019-06-20 20:00:00+00:00  FR04014       no2   21.4  µg/m³      6
```

By using `Timestamp` objects for dates, a lot of time-related properties are provided by pandas. For example the `month`, but also `year`, `weekofyear`, `quarter`,… All of these properties are accessible by the `dt` accessor.

An overview of the existing date properties is given in the *time and date components overview table*. More details about the `dt` accessor to return datetime like properties are explained in a dedicated section on the *dt accessor*.

What is the average NO_2 concentration for each day of the week for each of the measurement locations?

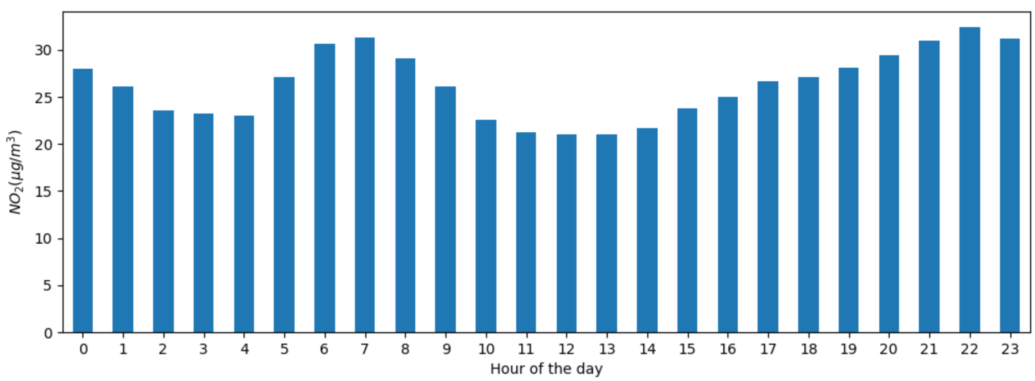
```
In [13]: air_quality.groupby(
   ....:     [air_quality["datetime"].dt.weekday, "location"])["value"].mean()
   ....:
Out[13]:
datetime  location
0         BETR801               27.875000
          FR04014               24.856250
          London Westminster    23.969697
1         BETR801               22.214286
          FR04014               30.999359
                                  ...
5         FR04014               25.266154
          London Westminster    24.977612
6         BETR801               21.896552
          FR04014               23.274306
          London Westminster    24.859155
Name: value, Length: 21, dtype: float64
```

Remember the split-apply-combine pattern provided by `groupby` from the *tutorial on statistics calculation*? Here, we want to calculate a given statistic (e.g. mean NO_2) **for each weekday** and **for each measurement location**. To group on weekdays, we use the datetime property `weekday` (with Monday=0 and Sunday=6) of pandas `Timestamp`, which is also accessible by the `dt` accessor. The grouping on both locations and weekdays can be done to split the calculation of the mean on each of these combinations.

Danger: As we are working with a very short time series in these examples, the analysis does not provide a long-term representative result!

Plot the typical NO_2 pattern during the day of our time series of all stations together. In other words, what is the average value for each hour of the day?

```
In [14]: fig, axs = plt.subplots(figsize=(12, 4))

In [15]: air_quality.groupby(air_quality["datetime"].dt.hour)["value"].mean().plot(
   ....:     kind='bar', rot=0, ax=axs
   ....: )
   ....:
Out[15]: <AxesSubplot:xlabel='datetime'>

In [16]: plt.xlabel("Hour of the day");  # custom x label using matplotlib

In [17]: plt.ylabel("$NO_2 (µg/m^3)$");
```



Similar to the previous case, we want to calculate a given statistic (e.g. mean NO_2) **for each hour of the day** and we can use the split-apply-combine approach again. For this case, we use the datetime property `hour` of pandas `Timestamp`, which is also accessible by the `dt` accessor.

Datetime as index

In the *tutorial on reshaping*, `pivot()` was introduced to reshape the data table with each of the measurements locations as a separate column:

```
In [18]: no_2 = air_quality.pivot(index="datetime", columns="location", values="value")

In [19]: no_2.head()
Out[19]:
location                   BETR801  FR04014  London Westminster
datetime
2019-05-07 01:00:00+00:00     50.5     25.0                23.0
2019-05-07 02:00:00+00:00     45.0     27.7                19.0
2019-05-07 03:00:00+00:00      NaN     50.4                19.0
2019-05-07 04:00:00+00:00      NaN     61.9                16.0
2019-05-07 05:00:00+00:00      NaN     72.4                 NaN
```

Note: By pivoting the data, the datetime information became the index of the table. In general, setting a column as an index can be achieved by the `set_index` function.

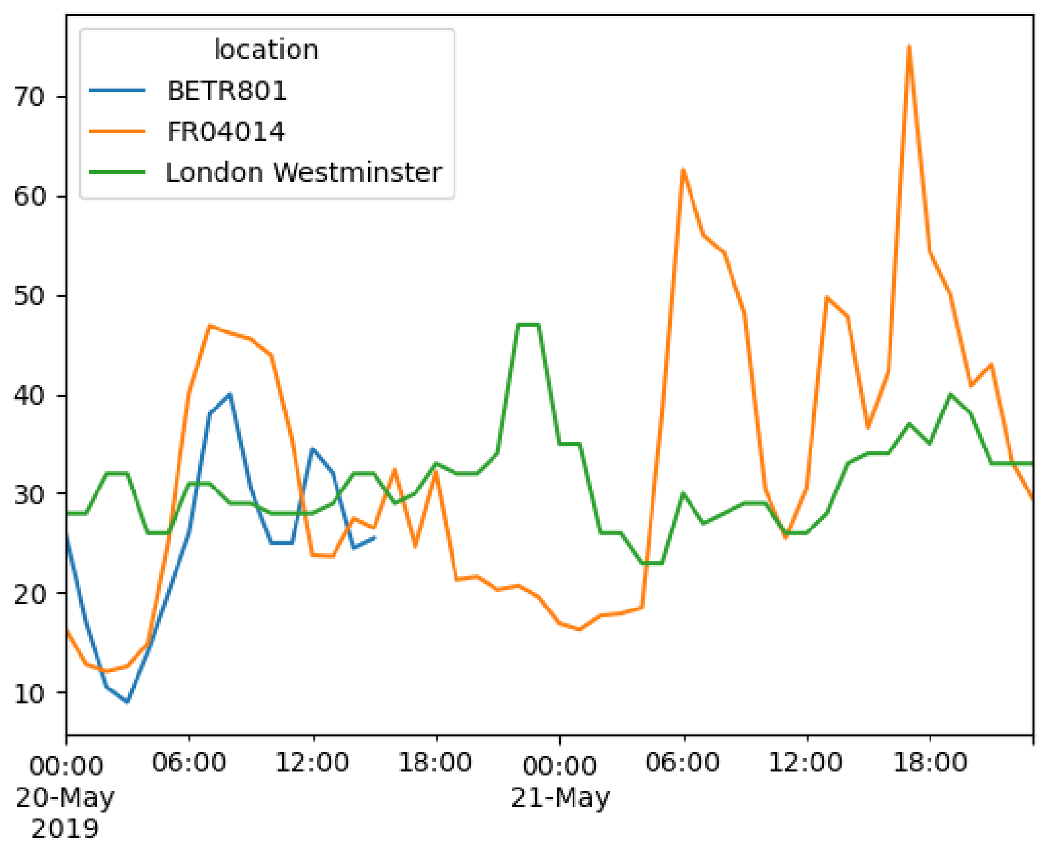
Working with a datetime index (i.e. `DatetimeIndex`) provides powerful functionalities. For example, we do not need the `dt` accessor to get the time series properties, but have these properties available on the index directly:

```
In [20]: no_2.index.year, no_2.index.weekday
Out[20]:
(Int64Index([2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019,
             ...
             2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019, 2019],
            dtype='int64', name='datetime', length=1033),
 Int64Index([1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
             ...
             3, 3, 3, 3, 3, 3, 3, 3, 3, 4],
            dtype='int64', name='datetime', length=1033))
```

Some other advantages are the convenient subsetting of time period or the adapted time scale on plots. Let's apply this on our data.

Create a plot of the NO_2 values in the different stations from the 20th of May till the end of 21st of May

```
In [21]: no_2["2019-05-20":"2019-05-21"].plot();
```



By providing a **string that parses to a datetime**, a specific subset of the data can be selected on a `DatetimeIndex`.

More information on the `DatetimeIndex` and the slicing by using strings is provided in the section on *time series indexing*.

Resample a time series to another frequency

Aggregate the current hourly time series values to the monthly maximum value in each of the stations.

```
In [22]: monthly_max = no_2.resample("M").max()

In [23]: monthly_max
Out[23]: 
location                   BETR801  FR04014  London Westminster
datetime                                                       
2019-05-31 00:00:00+00:00     74.5     97.0                97.0
2019-06-30 00:00:00+00:00     52.5     84.7                52.0
```

A very powerful method on time series data with a datetime index, is the ability to `resample()` time series to another frequency (e.g., converting secondly data into 5-minutely data).

The `resample()` method is similar to a groupby operation:

- it provides a time-based grouping, by using a string (e.g. `M`, `5H`,…) that defines the target frequency
- it requires an aggregation function such as `mean`, `max`,…

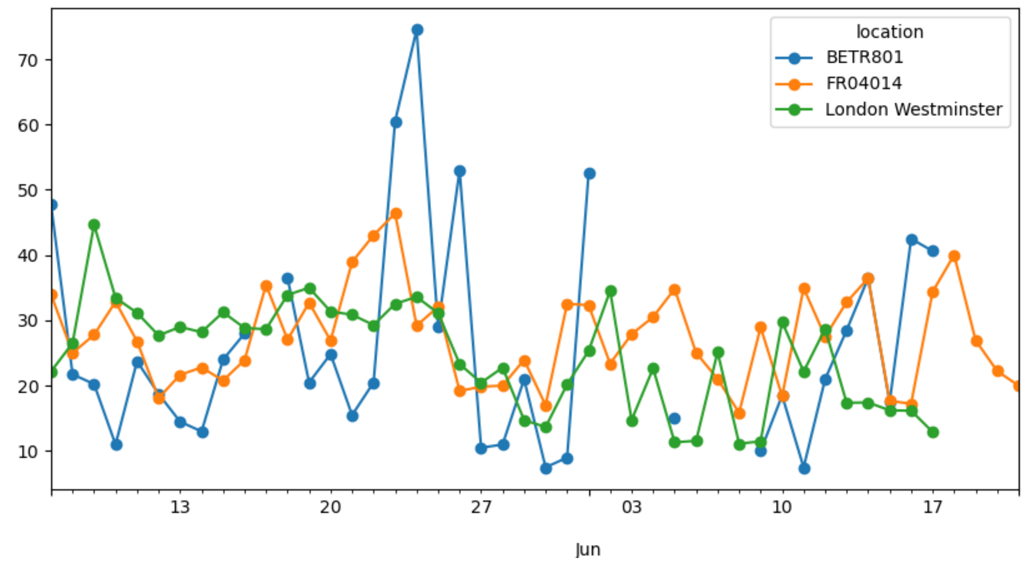
An overview of the aliases used to define time series frequencies is given in the *offset aliases overview table*.

When defined, the frequency of the time series is provided by the `freq` attribute:

```
In [24]: monthly_max.index.freq
Out[24]: <MonthEnd>
```

Make a plot of the daily mean NO_2 value in each of the stations.

```
In [25]: no_2.resample("D").mean().plot(style="-o", figsize=(10, 5));
```



More details on the power of time series `resampling` is provided in the user guide section on *resampling*.

- Valid date strings can be converted to datetime objects using `to_datetime` function or as part of read functions.
- Datetime objects in pandas support calculations, logical operations and convenient date-related properties using the `dt` accessor.
- A `DatetimeIndex` contains these date-related properties and supports convenient slicing.
- `Resample` is a powerful method to change the frequency of a time series.

A full overview on time series is given on the pages on *time series and date functionality*.

```
In [1]: import pandas as pd
```

This tutorial uses the Titanic data set, stored as CSV. The data consists of the following data columns:

- PassengerId: Id of every passenger.
- Survived: This feature have value 0 and 1. 0 for not survived and 1 for survived.
- Pclass: There are 3 classes: Class 1, Class 2 and Class 3.
- Name: Name of passenger.
- Sex: Gender of passenger.
- Age: Age of passenger.
- SibSp: Indication that passenger have siblings and spouse.
- Parch: Whether a passenger is alone or have family.
- Ticket: Ticket number of passenger.
- Fare: Indicating the fare.
- Cabin: The cabin of passenger.
- Embarked: The embarked category.

```
In [2]: titanic = pd.read_csv("data/titanic.csv")

In [3]: titanic.head()
Out[3]:
   PassengerId  Survived  Pclass                                               Name    ␣
↪Sex  ...  Parch            Ticket     Fare Cabin  Embarked
0            1         0       3                            Braund, Mr. Owen Harris   ␣
↪male  ...      0         A/5 21171   7.2500   NaN         S
1            2         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th... ␣
↪female  ...      0          PC 17599  71.2833   C85         C
2            3         1       3                             Heikkinen, Miss. Laina ␣
↪female  ...      0  STON/O2. 3101282   7.9250   NaN         S
3            4         1       1       Futrelle, Mrs. Jacques Heath (Lily May Peel) ␣
↪female  ...      0            113803  53.1000  C123         S
4            5         0       3                           Allen, Mr. William Henry   ␣
↪male  ...      0            373450   8.0500   NaN         S

[5 rows x 12 columns]
```

How to manipulate textual data?

Make all name characters lowercase.

```
In [4]: titanic["Name"].str.lower()
Out[4]:
0                                braund, mr. owen harris
1      cumings, mrs. john bradley (florence briggs th...
2                                 heikkinen, miss. laina
3           futrelle, mrs. jacques heath (lily may peel)
4                               allen, mr. william henry
                             ...
886                                montvila, rev. juozas
887                         graham, miss. margaret edith
888             johnston, miss. catherine helen "carrie"
889                                behr, mr. karl howell
890                                  dooley, mr. patrick
Name: Name, Length: 891, dtype: object
```

To make each of the strings in the `Name` column lowercase, select the `Name` column (see the *tutorial on selection of data*), add the `str` accessor and apply the `lower` method. As such, each of the strings is converted element-wise.

Similar to datetime objects in the *time series tutorial* having a `dt` accessor, a number of specialized string methods are available when using the `str` accessor. These methods have in general matching names with the equivalent built-in string methods for single elements, but are applied element-wise (remember *element-wise calculations*?) on each of the values of the columns.

Create a new column `Surname` that contains the surname of the passengers by extracting the part before the comma.

```
In [5]: titanic["Name"].str.split(",")
Out[5]:
0                             [Braund,  Mr. Owen Harris]
1      [Cumings,  Mrs. John Bradley (Florence Briggs ...
2                              [Heikkinen,  Miss. Laina]
```

(continues on next page)

(continued from previous page)

```
3        [Futrelle,  Mrs. Jacques Heath (Lily May Peel)]
4                            [Allen,  Mr. William Henry]
                                    ...
886                              [Montvila,  Rev. Juozas]
887                      [Graham,  Miss. Margaret Edith]
888          [Johnston,  Miss. Catherine Helen "Carrie"]
889                              [Behr,  Mr. Karl Howell]
890                                [Dooley,  Mr. Patrick]
Name: Name, Length: 891, dtype: object
```

Using the *Series.str.split()* method, each of the values is returned as a list of 2 elements. The first element is the part before the comma and the second element is the part after the comma.

```
In [6]: titanic["Surname"] = titanic["Name"].str.split(",").str.get(0)

In [7]: titanic["Surname"]
Out[7]: 
0         Braund
1        Cumings
2      Heikkinen
3       Futrelle
4          Allen
         ...    
886     Montvila
887       Graham
888     Johnston
889         Behr
890       Dooley
Name: Surname, Length: 891, dtype: object
```

As we are only interested in the first part representing the surname (element 0), we can again use the `str` accessor and apply *Series.str.get()* to extract the relevant part. Indeed, these string functions can be concatenated to combine multiple functions at once!

More information on extracting parts of strings is available in the user guide section on *splitting and replacing strings*.

Extract the passenger data about the countesses on board of the Titanic.

```
In [8]: titanic["Name"].str.contains("Countess")
Out[8]: 
0      False
1      False
2      False
3      False
4      False
       ...  
886    False
887    False
888    False
889    False
890    False
Name: Name, Length: 891, dtype: bool
```

```
In [9]: titanic[titanic["Name"].str.contains("Countess")]
Out[9]:
     PassengerId  Survived  Pclass                                               Name  ␣
↪ Sex   Age  ...  Parch  Ticket  Fare  Cabin Embarked Surname
759          760         1       1  Rothes, the Countess. of (Lucy Noel Martha Dye... ␣
↪female  33.0  ...      0  110152  86.5    B77        S  Rothes

[1 rows x 13 columns]
```

(*Interested in her story? See* Wikipedia*!*)

The string method *Series.str.contains()* checks for each of the values in the column `Name` if the string contains the word `Countess` and returns for each of the values `True` (`Countess` is part of the name) or `False` (`Countess` is not part of the name). This output can be used to subselect the data using conditional (boolean) indexing introduced in the *subsetting of data tutorial*. As there was only one countess on the Titanic, we get one row as a result.

Note: More powerful extractions on strings are supported, as the *Series.str.contains()* and *Series.str.extract()* methods accept regular expressions, but out of scope of this tutorial.

More information on extracting parts of strings is available in the user guide section on *string matching and extracting*.

Which passenger of the Titanic has the longest name?

```
In [10]: titanic["Name"].str.len()
Out[10]:
0      23
1      51
2      22
3      44
4      24
       ..
886    21
887    28
888    40
889    21
890    19
Name: Name, Length: 891, dtype: int64
```

To get the longest name we first have to get the lengths of each of the names in the `Name` column. By using pandas string methods, the *Series.str.len()* function is applied to each of the names individually (element-wise).

```
In [11]: titanic["Name"].str.len().idxmax()
Out[11]: 307
```

Next, we need to get the corresponding location, preferably the index label, in the table for which the name length is the largest. The *idxmax()* method does exactly that. It is not a string method and is applied to integers, so no `str` is used.

```
In [12]: titanic.loc[titanic["Name"].str.len().idxmax(), "Name"]
Out[12]: 'Penasco y Castellana, Mrs. Victor de Satode (Maria Josefa Perez de Soto y␣
↪Vallejo)'
```

Based on the index name of the row (`307`) and the column (`Name`), we can do a selection using the `loc` operator, introduced in the tutorial on subsetting.

In the "Sex" column, replace values of "male" by "M" and values of "female" by "F".

```
In [13]: titanic["Sex_short"] = titanic["Sex"].replace({"male": "M", "female": "F"})

In [14]: titanic["Sex_short"]
Out[14]:
0      M
1      F
2      F
3      F
4      M
      ..
886    M
887    F
888    F
889    M
890    M
Name: Sex_short, Length: 891, dtype: object
```

Whereas *replace()* is not a string method, it provides a convenient way to use mappings or vocabularies to translate certain values. It requires a `dictionary` to define the mapping `{from : to}`.

Warning: There is also a *replace()* method available to replace a specific set of characters. However, when having a mapping of multiple values, this would become:

```
titanic["Sex_short"] = titanic["Sex"].str.replace("female", "F")
titanic["Sex_short"] = titanic["Sex_short"].str.replace("male", "M")
```

This would become cumbersome and easily lead to mistakes. Just think (or try out yourself) what would happen if those two statements are applied in the opposite order…

- String methods are available using the `str` accessor.
- String methods work element-wise and can be used for conditional indexing.
- The `replace` method is a convenient method to convert values according to a given dictionary.

A full overview is provided in the user guide pages on *working with text data*.

1.4.4 Comparison with other tools

Comparison with R / R libraries

Since pandas aims to provide a lot of the data manipulation and analysis functionality that people use R for, this page was started to provide a more detailed look at the R language and its many third party libraries as they relate to pandas. In comparisons with R and CRAN libraries, we care about the following things:

- **Functionality / flexibility**: what can/cannot be done with each tool
- **Performance**: how fast are operations. Hard numbers/benchmarks are preferable
- **Ease-of-use**: Is one tool easier/harder to use (you may have to be the judge of this, given side-by-side code comparisons)

This page is also here to offer a bit of a translation guide for users of these R packages.

For transfer of `DataFrame` objects from pandas to R, one option is to use HDF5 files, see *External compatibility* for an example.

Quick reference

We'll start off with a quick reference guide pairing some common R operations using dplyr with pandas equivalents.

Querying, filtering, sampling

R	pandas
`dim(df)`	`df.shape`
`head(df)`	`df.head()`
`slice(df, 1:10)`	`df.iloc[:9]`
`filter(df, col1 == 1, col2 == 1)`	`df.query('col1 == 1 & col2 == 1')`
`df[df$col1 == 1 & df$col2 == 1,]`	`df[(df.col1 == 1) & (df.col2 == 1)]`
`select(df, col1, col2)`	`df[['col1', 'col2']]`
`select(df, col1:col3)`	`df.loc[:, 'col1':'col3']`
`select(df, -(col1:col3))`	`df.drop(cols_to_drop, axis=1)` but see[1]
`distinct(select(df, col1))`	`df[['col1']].drop_duplicates()`
`distinct(select(df, col1, col2))`	`df[['col1', 'col2']].drop_duplicates()`
`sample_n(df, 10)`	`df.sample(n=10)`
`sample_frac(df, 0.01)`	`df.sample(frac=0.01)`

Sorting

R	pandas
`arrange(df, col1, col2)`	`df.sort_values(['col1', 'col2'])`
`arrange(df, desc(col1))`	`df.sort_values('col1', ascending=False)`

Transforming

R	pandas
`select(df, col_one = col1)`	`df.rename(columns={'col1': 'col_one'})['col_one']`
`rename(df, col_one = col1)`	`df.rename(columns={'col1': 'col_one'})`
`mutate(df, c=a-b)`	`df.assign(c=df['a']-df['b'])`

[1] R's shorthand for a subrange of columns (`select(df, col1:col3)`) can be approached cleanly in pandas, if you have the list of columns, for example `df[cols[1:3]]` or `df.drop(cols[1:3])`, but doing this by column name is a bit messy.

Grouping and summarizing

R	pandas
`summary(df)`	`df.describe()`
`gdf <- group_by(df, col1)`	`gdf = df.groupby('col1')`
`summarise(gdf, avg=mean(col1, na.rm=TRUE))`	`df.groupby('col1').agg({'col1': 'mean'})`
`summarise(gdf, total=sum(col1))`	`df.groupby('col1').sum()`

Base R

Slicing with R's c

R makes it easy to access `data.frame` columns by name

```
df <- data.frame(a=rnorm(5), b=rnorm(5), c=rnorm(5), d=rnorm(5), e=rnorm(5))
df[, c("a", "c", "e")]
```

or by integer location

```
df <- data.frame(matrix(rnorm(1000), ncol=100))
df[, c(1:10, 25:30, 40, 50:100)]
```

Selecting multiple columns by name in pandas is straightforward

```
In [1]: df = pd.DataFrame(np.random.randn(10, 3), columns=list("abc"))

In [2]: df[["a", "c"]]
Out[2]:
          a         c
0  0.469112 -1.509059
1 -1.135632 -0.173215
2  0.119209 -0.861849
3 -2.104569  1.071804
4  0.721555 -1.039575
5  0.271860  0.567020
6  0.276232 -0.673690
7  0.113648  0.524988
8  0.404705 -1.715002
9 -1.039268 -1.157892

In [3]: df.loc[:, ["a", "c"]]
Out[3]:
          a         c
0  0.469112 -1.509059
1 -1.135632 -0.173215
2  0.119209 -0.861849
3 -2.104569  1.071804
4  0.721555 -1.039575
5  0.271860  0.567020
6  0.276232 -0.673690
7  0.113648  0.524988
```

(continues on next page)

(continued from previous page)

```
8  0.404705 -1.715002
9 -1.039268 -1.157892
```

Selecting multiple noncontiguous columns by integer location can be achieved with a combination of the `iloc` indexer attribute and `numpy.r_`.

```
In [4]: named = list("abcdefg")

In [5]: n = 30

In [6]: columns = named + np.arange(len(named), n).tolist()

In [7]: df = pd.DataFrame(np.random.randn(n, n), columns=columns)

In [8]: df.iloc[:, np.r_[:10, 24:30]]
Out[8]:
          a         b         c         d         e         f         g  ...         9 ␣
↪      24        25        26        27        28        29
0  -1.344312  0.844885  1.075770 -0.109050  1.643563 -1.469388  0.357021  ... -0.968914 -
↪1.170299 -0.226169  0.410835  0.813850  0.132003 -0.827317
1  -0.076467 -1.187678  1.130127 -1.436737 -1.413681  1.607920  1.024180  ... -2.211372 ␣
↪0.959726 -1.110336 -0.619976  0.149748 -0.732339  0.687738
2   0.176444  0.403310 -0.154951  0.301624 -2.179861 -1.369849 -0.954208  ... -0.826591 ␣
↪0.084844  0.432390  1.519970 -0.493662  0.600178  0.274230
3   0.132885 -0.023688  2.410179  1.450520  0.206053 -0.251905 -2.213588  ...  0.299368 -
↪2.484478 -0.281461  0.030711  0.109121  1.126203 -0.977349
4   1.474071 -0.064034 -1.282782  0.781836 -1.071357  0.441153  2.353925  ... -0.744471 -
↪1.197071 -1.066969 -0.303421 -0.858447  0.306996 -0.028665
..       ...       ...       ...       ...       ...       ...       ...  ...       ... ␣
↪     ...       ...       ...       ...       ...       ...
25  1.492125 -0.068190  0.681456  1.221829 -0.434352  1.204815 -0.195612  ... -0.796211 ␣
↪1.944517  0.042344 -0.307904  0.428572  0.880609  0.487645
26  0.725238  0.624607 -0.141185 -0.143948 -0.328162  2.095086 -0.608888  ... -2.513465 -
↪0.846188  1.190624  0.778507  1.008500  1.424017  0.717110
27  1.262419  1.950057  0.301038 -0.933858  0.814946  0.181439 -0.110015  ...  0.307941 -
↪1.341814  0.334281 -0.162227  1.007824  2.826008  1.458383
28 -1.585746 -0.899734  0.921494 -0.211762 -0.059182  0.058308  0.915377  ... -3.060395 ␣
↪0.403620 -0.026602 -0.240481  0.577223 -1.088417  0.326687
29 -0.986248  0.169729 -1.158091  1.019673  0.646039  0.917399 -0.010435  ...  0.869610 -
↪1.209247 -0.671466  0.332872 -2.013086 -1.602549  0.333109

[30 rows x 16 columns]
```

aggregate

In R you may want to split data into subsets and compute the mean for each. Using a data.frame called `df` and splitting it into groups `by1` and `by2`:

```
df <- data.frame(
  v1 = c(1,3,5,7,8,3,5,NA,4,5,7,9),
  v2 = c(11,33,55,77,88,33,55,NA,44,55,77,99),
  by1 = c("red", "blue", 1, 2, NA, "big", 1, 2, "red", 1, NA, 12),
  by2 = c("wet", "dry", 99, 95, NA, "damp", 95, 99, "red", 99, NA, NA))
aggregate(x=df[, c("v1", "v2")], by=list(mydf2$by1, mydf2$by2), FUN = mean)
```

The *groupby()* method is similar to base R `aggregate` function.

```
In [9]: df = pd.DataFrame(
   ...:     {
   ...:         "v1": [1, 3, 5, 7, 8, 3, 5, np.nan, 4, 5, 7, 9],
   ...:         "v2": [11, 33, 55, 77, 88, 33, 55, np.nan, 44, 55, 77, 99],
   ...:         "by1": ["red", "blue", 1, 2, np.nan, "big", 1, 2, "red", 1, np.nan, 12],
   ...:         "by2": [
   ...:             "wet",
   ...:             "dry",
   ...:             99,
   ...:             95,
   ...:             np.nan,
   ...:             "damp",
   ...:             95,
   ...:             99,
   ...:             "red",
   ...:             99,
   ...:             np.nan,
   ...:             np.nan,
   ...:         ],
   ...:     }
   ...: )
   ...:

In [10]: g = df.groupby(["by1", "by2"])

In [11]: g[["v1", "v2"]].mean()
Out[11]:
            v1    v2
by1  by2
1    95    5.0  55.0
     99    5.0  55.0
2    95    7.0  77.0
     99    NaN   NaN
big  damp  3.0  33.0
blue dry   3.0  33.0
red  red   4.0  44.0
     wet   1.0  11.0
```

For more details and examples see *the groupby documentation*.

match / %in%

A common way to select data in R is using `%in%` which is defined using the function `match`. The operator `%in%` is used to return a logical vector indicating if there is a match or not:

```
s <- 0:4
s %in% c(2,4)
```

The *isin()* method is similar to R `%in%` operator:

```
In [12]: s = pd.Series(np.arange(5), dtype=np.float32)

In [13]: s.isin([2, 4])
Out[13]: 
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

The `match` function returns a vector of the positions of matches of its first argument in its second:

```
s <- 0:4
match(s, c(2,4))
```

For more details and examples see *the reshaping documentation*.

tapply

`tapply` is similar to `aggregate`, but data can be in a ragged array, since the subclass sizes are possibly irregular. Using a data.frame called `baseball`, and retrieving information based on the array `team`:

```
baseball <-
  data.frame(team = gl(5, 5,
             labels = paste("Team", LETTERS[1:5])),
             player = sample(letters, 25),
             batting.average = runif(25, .200, .400))

tapply(baseball$batting.average, baseball.example$team,
       max)
```

In pandas we may use *pivot_table()* method to handle this:

```
In [14]: import random

In [15]: import string

In [16]: baseball = pd.DataFrame(
   ....:     {
   ....:         "team": ["team %d" % (x + 1) for x in range(5)] * 5,
   ....:         "player": random.sample(list(string.ascii_lowercase), 25),
   ....:         "batting avg": np.random.uniform(0.200, 0.400, 25),
```

(continues on next page)

(continued from previous page)

```
   ....:     }
   ....: )
   ....:

In [17]: baseball.pivot_table(values="batting avg", columns="team", aggfunc=np.max)
Out[17]:
team           team 1    team 2    team 3    team 4    team 5
batting avg  0.352134  0.295327  0.397191  0.394457  0.396194
```

For more details and examples see *the reshaping documentation*.

subset

The *query()* method is similar to the base R `subset` function. In R you might want to get the rows of a `data.frame` where one column's values are less than another column's values:

```
df <- data.frame(a=rnorm(10), b=rnorm(10))
subset(df, a <= b)
df[df$a <= df$b,]  # note the comma
```

In pandas, there are a few ways to perform subsetting. You can use *query()* or pass an expression as if it were an index/slice as well as standard boolean indexing:

```
In [18]: df = pd.DataFrame({"a": np.random.randn(10), "b": np.random.randn(10)})

In [19]: df.query("a <= b")
Out[19]:
          a         b
1  0.174950  0.552887
2 -0.023167  0.148084
3 -0.495291 -0.300218
4 -0.860736  0.197378
5 -1.134146  1.720780
7 -0.290098  0.083515
8  0.238636  0.946550

In [20]: df[df["a"] <= df["b"]]
Out[20]:
          a         b
1  0.174950  0.552887
2 -0.023167  0.148084
3 -0.495291 -0.300218
4 -0.860736  0.197378
5 -1.134146  1.720780
7 -0.290098  0.083515
8  0.238636  0.946550

In [21]: df.loc[df["a"] <= df["b"]]
Out[21]:
          a         b
1  0.174950  0.552887
```

(continues on next page)

(continued from previous page)

```
2 -0.023167  0.148084
3 -0.495291 -0.300218
4 -0.860736  0.197378
5 -1.134146  1.720780
7 -0.290098  0.083515
8  0.238636  0.946550
```

For more details and examples see *the query documentation*.

with

An expression using a data.frame called `df` in R with the columns `a` and `b` would be evaluated using `with` like so:

```
df <- data.frame(a=rnorm(10), b=rnorm(10))
with(df, a + b)
df$a + df$b  # same as the previous expression
```

In pandas the equivalent expression, using the `eval()` method, would be:

```
In [22]: df = pd.DataFrame({"a": np.random.randn(10), "b": np.random.randn(10)})

In [23]: df.eval("a + b")
Out[23]:
0   -0.091430
1   -2.483890
2   -0.252728
3   -0.626444
4   -0.261740
5    2.149503
6   -0.332214
7    0.799331
8   -2.377245
9    2.104677
dtype: float64

In [24]: df["a"] + df["b"]  # same as the previous expression
Out[24]:
0   -0.091430
1   -2.483890
2   -0.252728
3   -0.626444
4   -0.261740
5    2.149503
6   -0.332214
7    0.799331
8   -2.377245
9    2.104677
dtype: float64
```

In certain cases `eval()` will be much faster than evaluation in pure Python. For more details and examples see *the eval documentation*.

plyr

plyr is an R library for the split-apply-combine strategy for data analysis. The functions revolve around three data structures in R, a for arrays, l for lists, and d for data.frame. The table below shows how these data structures could be mapped in Python.

R	Python
array	list
lists	dictionary or list of objects
data.frame	dataframe

ddply

An expression using a data.frame called df in R where you want to summarize x by month:

```
require(plyr)
df <- data.frame(
  x = runif(120, 1, 168),
  y = runif(120, 7, 334),
  z = runif(120, 1.7, 20.7),
  month = rep(c(5,6,7,8),30),
  week = sample(1:4, 120, TRUE)
)

ddply(df, .(month, week), summarize,
      mean = round(mean(x), 2),
      sd = round(sd(x), 2))
```

In pandas the equivalent expression, using the *groupby()* method, would be:

```
In [25]: df = pd.DataFrame(
   ....:     {
   ....:         "x": np.random.uniform(1.0, 168.0, 120),
   ....:         "y": np.random.uniform(7.0, 334.0, 120),
   ....:         "z": np.random.uniform(1.7, 20.7, 120),
   ....:         "month": [5, 6, 7, 8] * 30,
   ....:         "week": np.random.randint(1, 4, 120),
   ....:     }
   ....: )
   ....:

In [26]: grouped = df.groupby(["month", "week"])

In [27]: grouped["x"].agg([np.mean, np.std])
Out[27]:
                 mean        std
month week
5     1     63.653367  40.601965
      2     78.126605  53.342400
      3     92.091886  57.630110
6     1     81.747070  54.339218
      2     70.971205  54.687287
```

(continues on next page)

(continued from previous page)

```
      3     100.968344  54.010081
7     1      61.576332  38.844274
      2      61.733510  48.209013
      3      71.688795  37.595638
8     1      62.741922  34.618153
      2      91.774627  49.790202
      3      73.936856  60.773900
```

For more details and examples see *the groupby documentation*.

reshape / reshape2

meltarray

An expression using a 3 dimensional array called a in R where you want to melt it into a data.frame:

```
a <- array(c(1:23, NA), c(2,3,4))
data.frame(melt(a))
```

In Python, since a is a list, you can simply use list comprehension.

```
In [28]: a = np.array(list(range(1, 24)) + [np.NAN]).reshape(2, 3, 4)

In [29]: pd.DataFrame([tuple(list(x) + [val]) for x, val in np.ndenumerate(a)])
Out[29]:
    0  1  2     3
0   0  0  0   1.0
1   0  0  1   2.0
2   0  0  2   3.0
3   0  0  3   4.0
4   0  1  0   5.0
.. .. .. ..   ...
19  1  1  3  20.0
20  1  2  0  21.0
21  1  2  1  22.0
22  1  2  2  23.0
23  1  2  3   NaN

[24 rows x 4 columns]
```

meltlist

An expression using a list called a in R where you want to melt it into a data.frame:

```
a <- as.list(c(1:4, NA))
data.frame(melt(a))
```

In Python, this list would be a list of tuples, so `DataFrame()` method would convert it to a dataframe as required.

```
In [30]: a = list(enumerate(list(range(1, 5)) + [np.NAN]))

In [31]: pd.DataFrame(a)
Out[31]:
   0    1
0  0  1.0
1  1  2.0
2  2  3.0
3  3  4.0
4  4  NaN
```

For more details and examples see *the Into to Data Structures documentation*.

meltdf

An expression using a data.frame called cheese in R where you want to reshape the data.frame:

```
cheese <- data.frame(
  first = c('John', 'Mary'),
  last = c('Doe', 'Bo'),
  height = c(5.5, 6.0),
  weight = c(130, 150)
)
melt(cheese, id=c("first", "last"))
```

In Python, the *melt()* method is the R equivalent:

```
In [32]: cheese = pd.DataFrame(
   ....:     {
   ....:         "first": ["John", "Mary"],
   ....:         "last": ["Doe", "Bo"],
   ....:         "height": [5.5, 6.0],
   ....:         "weight": [130, 150],
   ....:     }
   ....: )
   ....:

In [33]: pd.melt(cheese, id_vars=["first", "last"])
Out[33]:
  first last variable  value
0  John  Doe   height    5.5
1  Mary   Bo   height    6.0
2  John  Doe   weight  130.0
3  Mary   Bo   weight  150.0

In [34]: cheese.set_index(["first", "last"]).stack()  # alternative way
Out[34]:
first  last
John   Doe   height      5.5
             weight    130.0
Mary   Bo    height      6.0
             weight    150.0
```

(continues on next page)

(continued from previous page)

```
dtype: float64
```

For more details and examples see *the reshaping documentation*.

cast

In R `acast` is an expression using a data.frame called `df` in R to cast into a higher dimensional array:

```
df <- data.frame(
  x = runif(12, 1, 168),
  y = runif(12, 7, 334),
  z = runif(12, 1.7, 20.7),
  month = rep(c(5,6,7),4),
  week = rep(c(1,2), 6)
)

mdf <- melt(df, id=c("month", "week"))
acast(mdf, week ~ month ~ variable, mean)
```

In Python the best way is to make use of `pivot_table()`:

```
In [35]: df = pd.DataFrame(
   ....:     {
   ....:         "x": np.random.uniform(1.0, 168.0, 12),
   ....:         "y": np.random.uniform(7.0, 334.0, 12),
   ....:         "z": np.random.uniform(1.7, 20.7, 12),
   ....:         "month": [5, 6, 7] * 4,
   ....:         "week": [1, 2] * 6,
   ....:     }
   ....: )
   ....:

In [36]: mdf = pd.melt(df, id_vars=["month", "week"])

In [37]: pd.pivot_table(
   ....:     mdf,
   ....:     values="value",
   ....:     index=["variable", "week"],
   ....:     columns=["month"],
   ....:     aggfunc=np.mean,
   ....: )
   ....:
Out[37]:
month                  5           6           7
variable week
x        1     93.888747   98.762034   55.219673
         2     94.391427   38.112932   83.942781
y        1     94.306912  279.454811  227.840449
         2     87.392662  193.028166  173.899260
z        1     11.016009   10.079307   16.170549
         2      8.476111   17.638509   19.003494
```

Similarly for `dcast` which uses a data.frame called `df` in R to aggregate information based on `Animal` and `FeedType`:

```
df <- data.frame(
  Animal = c('Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
             'Animal2', 'Animal3'),
  FeedType = c('A', 'B', 'A', 'A', 'B', 'B', 'A'),
  Amount = c(10, 7, 4, 2, 5, 6, 2)
)

dcast(df, Animal ~ FeedType, sum, fill=NaN)
# Alternative method using base R
with(df, tapply(Amount, list(Animal, FeedType), sum))
```

Python can approach this in two different ways. Firstly, similar to above using *pivot_table()*:

```
In [38]: df = pd.DataFrame(
   ....:     {
   ....:         "Animal": [
   ....:             "Animal1",
   ....:             "Animal2",
   ....:             "Animal3",
   ....:             "Animal2",
   ....:             "Animal1",
   ....:             "Animal2",
   ....:             "Animal3",
   ....:         ],
   ....:         "FeedType": ["A", "B", "A", "A", "B", "B", "A"],
   ....:         "Amount": [10, 7, 4, 2, 5, 6, 2],
   ....:     }
   ....: )
   ....:

In [39]: df.pivot_table(values="Amount", index="Animal", columns="FeedType", aggfunc="sum
↪")
Out[39]:
FeedType     A     B
Animal
Animal1   10.0   5.0
Animal2    2.0  13.0
Animal3    6.0   NaN
```

The second approach is to use the *groupby()* method:

```
In [40]: df.groupby(["Animal", "FeedType"])["Amount"].sum()
Out[40]:
Animal   FeedType
Animal1  A           10
         B            5
Animal2  A            2
         B           13
Animal3  A            6
Name: Amount, dtype: int64
```

For more details and examples see *the reshaping documentation* or *the groupby documentation*.

factor

pandas has a data type for categorical data.

```
cut(c(1,2,3,4,5,6), 3)
factor(c(1,2,3,2,2,3))
```

In pandas this is accomplished with `pd.cut` and `astype("category")`:

```
In [41]: pd.cut(pd.Series([1, 2, 3, 4, 5, 6]), 3)
Out[41]:
0    (0.995, 2.667]
1    (0.995, 2.667]
2    (2.667, 4.333]
3    (2.667, 4.333]
4      (4.333, 6.0]
5      (4.333, 6.0]
dtype: category
Categories (3, interval[float64, right]): [(0.995, 2.667] < (2.667, 4.333] < (4.333, 6.
↪0]]

In [42]: pd.Series([1, 2, 3, 2, 2, 3]).astype("category")
Out[42]:
0    1
1    2
2    3
3    2
4    2
5    3
dtype: category
Categories (3, int64): [1, 2, 3]
```

For more details and examples see *categorical introduction* and the *API documentation*. There is also a documentation regarding the *differences to R's factor*.

Comparison with SQL

Since many potential pandas users have some familiarity with SQL, this page is meant to provide some examples of how various SQL operations would be performed using pandas.

If you're new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

```
In [1]: import pandas as pd

In [2]: import numpy as np
```

Most of the examples will utilize the `tips` dataset found within pandas tests. We'll read the data into a DataFrame called `tips` and assume we have a database table of the same name and structure.

```
In [3]: url = (
   ...:     "https://raw.github.com/pandas-dev"
```

(continues on next page)

(continued from previous page)

```
   ...:     "/pandas/main/pandas/tests/io/data/csv/tips.csv"
   ...: )
   ...:

In [4]: tips = pd.read_csv(url)

In [5]: tips
Out[5]:
     total_bill   tip     sex smoker   day    time  size
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
4         24.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       29.03  5.92    Male     No   Sat  Dinner     3
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]
```

Copies vs. in place operations

Most pandas operations return copies of the `Series`/`DataFrame`. To make the changes "stick", you'll need to either assign to a new variable:

```
sorted_df = df.sort_values("col1")
```

or overwrite the original one:

```
df = df.sort_values("col1")
```

Note: You will see an `inplace=True` keyword argument available for some methods:

```
df.sort_values("col1", inplace=True)
```

Its use is discouraged. *More information.*

SELECT

In SQL, selection is done using a comma-separated list of columns you'd like to select (or a * to select all columns):

```
SELECT total_bill, tip, smoker, time
FROM tips;
```

With pandas, column selection is done by passing a list of column names to your DataFrame:

```
In [6]: tips[["total_bill", "tip", "smoker", "time"]]
Out[6]:
     total_bill   tip smoker    time
0         16.99  1.01     No  Dinner
1         10.34  1.66     No  Dinner
2         21.01  3.50     No  Dinner
3         23.68  3.31     No  Dinner
4         24.59  3.61     No  Dinner
..          ...   ...    ...     ...
239       29.03  5.92     No  Dinner
240       27.18  2.00    Yes  Dinner
241       22.67  2.00    Yes  Dinner
242       17.82  1.75     No  Dinner
243       18.78  3.00     No  Dinner

[244 rows x 4 columns]
```

Calling the DataFrame without the list of column names would display all columns (akin to SQL's *).

In SQL, you can add a calculated column:

```
SELECT *, tip/total_bill as tip_rate
FROM tips;
```

With pandas, you can use the `DataFrame.assign()` method of a DataFrame to append a new column:

```
In [7]: tips.assign(tip_rate=tips["tip"] / tips["total_bill"])
Out[7]:
     total_bill   tip     sex smoker   day    time  size  tip_rate
0         16.99  1.01  Female     No   Sun  Dinner     2  0.059447
1         10.34  1.66    Male     No   Sun  Dinner     3  0.160542
2         21.01  3.50    Male     No   Sun  Dinner     3  0.166587
3         23.68  3.31    Male     No   Sun  Dinner     2  0.139780
4         24.59  3.61  Female     No   Sun  Dinner     4  0.146808
..          ...   ...     ...    ...   ...     ...   ...       ...
239       29.03  5.92    Male     No   Sat  Dinner     3  0.203927
240       27.18  2.00  Female    Yes   Sat  Dinner     2  0.073584
241       22.67  2.00    Male    Yes   Sat  Dinner     2  0.088222
242       17.82  1.75    Male     No   Sat  Dinner     2  0.098204
243       18.78  3.00  Female     No  Thur  Dinner     2  0.159744

[244 rows x 8 columns]
```

WHERE

Filtering in SQL is done via a WHERE clause.

```
SELECT *
FROM tips
WHERE time = 'Dinner';
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using *boolean indexing*.

```
In [8]: tips[tips["total_bill"] > 10]
Out[8]:
     total_bill   tip     sex smoker   day    time  size
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
4         24.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       29.03  5.92    Male     No   Sat  Dinner     3
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2

[227 rows x 7 columns]
```

The above statement is simply passing a `Series` of `True`/`False` objects to the DataFrame, returning all rows with `True`.

```
In [9]: is_dinner = tips["time"] == "Dinner"

In [10]: is_dinner
Out[10]:
0       True
1       True
2       True
3       True
4       True
        ...
239     True
240     True
241     True
242     True
243     True
Name: time, Length: 244, dtype: bool

In [11]: is_dinner.value_counts()
Out[11]:
True     176
False     68
Name: time, dtype: int64
```

(continues on next page)

(continued from previous page)

```
In [12]: tips[is_dinner]
Out[12]:
     total_bill   tip     sex smoker   day    time  size
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
4         24.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       29.03  5.92    Male     No   Sat  Dinner     3
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2

[176 rows x 7 columns]
```

Just like SQL's OR and AND, multiple conditions can be passed to a DataFrame using | (OR) and & (AND).

Tips of more than $5 at Dinner meals:

```
SELECT *
FROM tips
WHERE time = 'Dinner' AND tip > 5.00;
```

```
In [13]: tips[(tips["time"] == "Dinner") & (tips["tip"] > 5.00)]
Out[13]:
     total_bill    tip     sex smoker  day    time  size
23        39.42   7.58    Male     No  Sat  Dinner     4
44        30.40   5.60    Male     No  Sun  Dinner     4
47        32.40   6.00    Male     No  Sun  Dinner     4
52        34.81   5.20  Female     No  Sun  Dinner     4
59        48.27   6.73    Male     No  Sat  Dinner     4
116       29.93   5.07    Male     No  Sun  Dinner     4
155       29.85   5.14  Female     No  Sun  Dinner     5
170       50.81  10.00    Male    Yes  Sat  Dinner     3
172        7.25   5.15    Male    Yes  Sun  Dinner     2
181       23.33   5.65    Male    Yes  Sun  Dinner     2
183       23.17   6.50    Male    Yes  Sun  Dinner     4
211       25.89   5.16    Male    Yes  Sat  Dinner     4
212       48.33   9.00    Male     No  Sat  Dinner     4
214       28.17   6.50  Female    Yes  Sat  Dinner     3
239       29.03   5.92    Male     No  Sat  Dinner     3
```

Tips by parties of at least 5 diners OR bill total was more than $45:

```
SELECT *
FROM tips
WHERE size >= 5 OR total_bill > 45;
```

```
In [14]: tips[(tips["size"] >= 5) | (tips["total_bill"] > 45)]
Out[14]:
```

(continues on next page)

(continued from previous page)

```
     total_bill    tip     sex smoker   day    time  size
59        48.27   6.73    Male     No   Sat  Dinner     4
125       29.80   4.20  Female     No  Thur   Lunch     6
141       34.30   6.70    Male     No  Thur   Lunch     6
142       41.19   5.00    Male     No  Thur   Lunch     5
143       27.05   5.00  Female     No  Thur   Lunch     6
155       29.85   5.14  Female     No   Sun  Dinner     5
156       48.17   5.00    Male     No   Sun  Dinner     6
170       50.81  10.00    Male    Yes   Sat  Dinner     3
182       45.35   3.50    Male    Yes   Sun  Dinner     3
185       20.69   5.00    Male     No   Sun  Dinner     5
187       30.46   2.00    Male    Yes   Sun  Dinner     5
212       48.33   9.00    Male     No   Sat  Dinner     4
216       28.15   3.00    Male    Yes   Sat  Dinner     5
```

NULL checking is done using the *notna()* and *isna()* methods.

```
In [15]: frame = pd.DataFrame(
   ....:     {"col1": ["A", "B", np.NaN, "C", "D"], "col2": ["F", np.NaN, "G", "H", "I"]}
   ....: )
   ....:

In [16]: frame
Out[16]:
  col1 col2
0    A    F
1    B  NaN
2  NaN    G
3    C    H
4    D    I
```

Assume we have a table of the same structure as our DataFrame above. We can see only the records where `col2` IS NULL with the following query:

```
SELECT *
FROM frame
WHERE col2 IS NULL;
```

```
In [17]: frame[frame["col2"].isna()]
Out[17]:
  col1 col2
1    B  NaN
```

Getting items where `col1` IS NOT NULL can be done with *notna()*.

```
SELECT *
FROM frame
WHERE col1 IS NOT NULL;
```

```
In [18]: frame[frame["col1"].notna()]
Out[18]:
  col1 col2
```

(continues on next page)

(continued from previous page)

```
0    A    F
1    B  NaN
3    C    H
4    D    I
```

GROUP BY

In pandas, SQL's `GROUP BY` operations are performed using the similarly named *groupby()* method. *groupby()* typically refers to a process where we'd like to split a dataset into groups, apply some function (typically aggregation) , and then combine the groups together.

A common SQL operation would be getting the count of records in each group throughout a dataset. For instance, a query getting us the number of tips left by sex:

```
SELECT sex, count(*)
FROM tips
GROUP BY sex;
/*
Female     87
Male      157
*/
```

The pandas equivalent would be:

```
In [19]: tips.groupby("sex").size()
Out[19]:
sex
Female     87
Male      157
dtype: int64
```

Notice that in the pandas code we used *size()* and not *count()*. This is because *count()* applies the function to each column, returning the number of `NOT NULL` records within each.

```
In [20]: tips.groupby("sex").count()
Out[20]:
        total_bill  tip  smoker  day  time  size
sex
Female          87   87      87   87    87    87
Male           157  157     157  157   157   157
```

Alternatively, we could have applied the *count()* method to an individual column:

```
In [21]: tips.groupby("sex")["total_bill"].count()
Out[21]:
sex
Female     87
Male      157
Name: total_bill, dtype: int64
```

Multiple functions can also be applied at once. For instance, say we'd like to see how tip amount differs by day of the week - `agg()` allows you to pass a dictionary to your grouped DataFrame, indicating which functions to apply to specific columns.

```sql
SELECT day, AVG(tip), COUNT(*)
FROM tips
GROUP BY day;
/*
Fri   2.734737   19
Sat   2.993103   87
Sun   3.255132   76
Thu  2.771452   62
*/
```

```
In [22]: tips.groupby("day").agg({"tip": np.mean, "day": np.size})
Out[22]:
           tip  day
day
Fri   2.734737   19
Sat   2.993103   87
Sun   3.255132   76
Thur  2.771452   62
```

Grouping by more than one column is done by passing a list of columns to the `groupby()` method.

```sql
SELECT smoker, day, COUNT(*), AVG(tip)
FROM tips
GROUP BY smoker, day;
/*
smoker day
No     Fri      4   2.812500
       Sat     45   3.102889
       Sun     57   3.167895
       Thu    45   2.673778
Yes    Fri     15   2.714000
       Sat     42   2.875476
       Sun     19   3.516842
       Thu    17   3.030000
*/
```

```
In [23]: tips.groupby(["smoker", "day"]).agg({"tip": [np.size, np.mean]})
Out[23]:
             tip
            size      mean
smoker day
No     Fri     4  2.812500
       Sat    45  3.102889
       Sun    57  3.167895
       Thur   45  2.673778
Yes    Fri    15  2.714000
       Sat    42  2.875476
       Sun    19  3.516842
       Thur   17  3.030000
```

JOIN

JOINs can be performed with *join()* or *merge()*. By default, *join()* will join the DataFrames on their indices. Each method has parameters allowing you to specify the type of join to perform (LEFT, RIGHT, INNER, FULL) or the columns to join on (column names or indices).

> **Warning:** If both key columns contain rows where the key is a null value, those rows will be matched against each other. This is different from usual SQL join behaviour and can lead to unexpected results.

```
In [24]: df1 = pd.DataFrame({"key": ["A", "B", "C", "D"], "value": np.random.randn(4)})

In [25]: df2 = pd.DataFrame({"key": ["B", "D", "D", "E"], "value": np.random.randn(4)})
```

Assume we have two database tables of the same name and structure as our DataFrames.

Now let's go over the various types of JOINs.

INNER JOIN

```
SELECT *
FROM df1
INNER JOIN df2
  ON df1.key = df2.key;
```

```
# merge performs an INNER JOIN by default
In [26]: pd.merge(df1, df2, on="key")
Out[26]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209
```

merge() also offers parameters for cases when you'd like to join one DataFrame's column with another DataFrame's index.

```
In [27]: indexed_df2 = df2.set_index("key")

In [28]: pd.merge(df1, indexed_df2, left_on="key", right_index=True)
Out[28]:
  key   value_x   value_y
1   B -0.282863  1.212112
3   D -1.135632 -0.173215
3   D -1.135632  0.119209
```

LEFT OUTER JOIN

Show all records from df1.

```
SELECT *
FROM df1
LEFT OUTER JOIN df2
  ON df1.key = df2.key;
```

```
In [29]: pd.merge(df1, df2, on="key", how="left")
Out[29]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
```

RIGHT JOIN

Show all records from df2.

```
SELECT *
FROM df1
RIGHT OUTER JOIN df2
  ON df1.key = df2.key;
```

```
In [30]: pd.merge(df1, df2, on="key", how="right")
Out[30]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209
3   E       NaN -1.044236
```

FULL JOIN

pandas also allows for FULL JOINs, which display both sides of the dataset, whether or not the joined columns find a match. As of writing, FULL JOINs are not supported in all RDBMS (MySQL).

Show all records from both tables.

```
SELECT *
FROM df1
FULL OUTER JOIN df2
  ON df1.key = df2.key;
```

```
In [31]: pd.merge(df1, df2, on="key", how="outer")
Out[31]:
  key   value_x   value_y
```

(continues on next page)

(continued from previous page)

```
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236
```

UNION

UNION ALL can be performed using *concat()*.

```
In [32]: df1 = pd.DataFrame(
   ....:     {"city": ["Chicago", "San Francisco", "New York City"], "rank": range(1, 4)}
   ....: )
   ....:

In [33]: df2 = pd.DataFrame(
   ....:     {"city": ["Chicago", "Boston", "Los Angeles"], "rank": [1, 4, 5]}
   ....: )
   ....:
```

```
SELECT city, rank
FROM df1
UNION ALL
SELECT city, rank
FROM df2;
/*
         city  rank
      Chicago     1
San Francisco     2
New York City     3
      Chicago     1
       Boston     4
  Los Angeles     5
*/
```

```
In [34]: pd.concat([df1, df2])
Out[34]:
            city  rank
0        Chicago     1
1  San Francisco     2
2  New York City     3
0        Chicago     1
1         Boston     4
2    Los Angeles     5
```

SQL's UNION is similar to UNION ALL, however UNION will remove duplicate rows.

```
SELECT city, rank
FROM df1
```

(continues on next page)

(continued from previous page)

```
UNION
SELECT city, rank
FROM df2;
-- notice that there is only one Chicago record this time
/*
         city  rank
      Chicago     1
San Francisco     2
New York City     3
       Boston     4
  Los Angeles     5
*/
```

In pandas, you can use `concat()` in conjunction with `drop_duplicates()`.

```
In [35]: pd.concat([df1, df2]).drop_duplicates()
Out[35]:
            city  rank
0        Chicago     1
1  San Francisco     2
2  New York City     3
1         Boston     4
2    Los Angeles     5
```

LIMIT

```
SELECT * FROM tips
LIMIT 10;
```

```
In [36]: tips.head(10)
Out[36]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
5       25.29  4.71    Male     No  Sun  Dinner     4
6        8.77  2.00    Male     No  Sun  Dinner     2
7       26.88  3.12    Male     No  Sun  Dinner     4
8       15.04  1.96    Male     No  Sun  Dinner     2
9       14.78  3.23    Male     No  Sun  Dinner     2
```

pandas equivalents for some SQL analytic and aggregate functions

Top n rows with offset

```
-- MySQL
SELECT * FROM tips
ORDER BY tip DESC
LIMIT 10 OFFSET 5;
```

```
In [37]: tips.nlargest(10 + 5, columns="tip").tail(10)
Out[37]:
     total_bill   tip     sex smoker   day    time  size
183       23.17  6.50    Male    Yes   Sun  Dinner     4
214       28.17  6.50  Female    Yes   Sat  Dinner     3
47        32.40  6.00    Male     No   Sun  Dinner     4
239       29.03  5.92    Male     No   Sat  Dinner     3
88        24.71  5.85    Male     No  Thur   Lunch     2
181       23.33  5.65    Male    Yes   Sun  Dinner     2
44        30.40  5.60    Male     No   Sun  Dinner     4
52        34.81  5.20  Female     No   Sun  Dinner     4
85        34.83  5.17  Female     No  Thur   Lunch     4
211       25.89  5.16    Male    Yes   Sat  Dinner     4
```

Top n rows per group

```
-- Oracle's ROW_NUMBER() analytic function
SELECT * FROM (
  SELECT
    t.*,
    ROW_NUMBER() OVER(PARTITION BY day ORDER BY total_bill DESC) AS rn
  FROM tips t
)
WHERE rn < 3
ORDER BY day, rn;
```

```
In [38]: (
   ....:     tips.assign(
   ....:         rn=tips.sort_values(["total_bill"], ascending=False)
   ....:         .groupby(["day"])
   ....:         .cumcount()
   ....:         + 1
   ....:     )
   ....:     .query("rn < 3")
   ....:     .sort_values(["day", "rn"])
   ....: )
   ....:
Out[38]:
     total_bill    tip     sex smoker   day    time  size  rn
95        40.17   4.73    Male    Yes   Fri  Dinner     4   1
90        28.97   3.00    Male    Yes   Fri  Dinner     2   2
```

(continues on next page)

(continued from previous page)

```
170       50.81  10.00    Male    Yes   Sat  Dinner     3    1
212       48.33   9.00    Male     No   Sat  Dinner     4    2
156       48.17   5.00    Male     No   Sun  Dinner     6    1
182       45.35   3.50    Male    Yes   Sun  Dinner     3    2
197       43.11   5.00  Female    Yes  Thur   Lunch     4    1
142       41.19   5.00    Male     No  Thur   Lunch     5    2
```

the same using `rank(method='first')` function

```
In [39]: (
   ....:     tips.assign(
   ....:         rnk=tips.groupby(["day"])["total_bill"].rank(
   ....:             method="first", ascending=False
   ....:         )
   ....:     )
   ....:     .query("rnk < 3")
   ....:     .sort_values(["day", "rnk"])
   ....: )
   ....:
Out[39]:
     total_bill    tip     sex smoker   day    time  size  rnk
95        40.17   4.73    Male    Yes   Fri  Dinner     4  1.0
90        28.97   3.00    Male    Yes   Fri  Dinner     2  2.0
170       50.81  10.00    Male    Yes   Sat  Dinner     3  1.0
212       48.33   9.00    Male     No   Sat  Dinner     4  2.0
156       48.17   5.00    Male     No   Sun  Dinner     6  1.0
182       45.35   3.50    Male    Yes   Sun  Dinner     3  2.0
197       43.11   5.00  Female    Yes  Thur   Lunch     4  1.0
142       41.19   5.00    Male     No  Thur   Lunch     5  2.0
```

```
-- Oracle's RANK() analytic function
SELECT * FROM (
  SELECT
    t.*,
    RANK() OVER(PARTITION BY sex ORDER BY tip) AS rnk
  FROM tips t
  WHERE tip < 2
)
WHERE rnk < 3
ORDER BY sex, rnk;
```

Let's find tips with (rank < 3) per gender group for (tips < 2). Notice that when using `rank(method='min')` function `rnk_min` remains the same for the same `tip` (as Oracle's `RANK()` function)

```
In [40]: (
   ....:     tips[tips["tip"] < 2]
   ....:     .assign(rnk_min=tips.groupby(["sex"])["tip"].rank(method="min"))
   ....:     .query("rnk_min < 3")
   ....:     .sort_values(["sex", "rnk_min"])
   ....: )
   ....:
Out[40]:
```

(continues on next page)

(continued from previous page)

```
     total_bill   tip     sex smoker  day    time  size  rnk_min
67         3.07  1.00  Female    Yes  Sat  Dinner     1      1.0
92         5.75  1.00  Female    Yes  Fri  Dinner     2      1.0
111        7.25  1.00  Female     No  Sat  Dinner     1      1.0
236       12.60  1.00    Male    Yes  Sat  Dinner     2      1.0
237       32.83  1.17    Male    Yes  Sat  Dinner     2      2.0
```

UPDATE

```
UPDATE tips
SET tip = tip*2
WHERE tip < 2;
```

```
In [41]: tips.loc[tips["tip"] < 2, "tip"] *= 2
```

DELETE

```
DELETE FROM tips
WHERE tip > 9;
```

In pandas we select the rows that should remain instead of deleting them:

```
In [42]: tips = tips.loc[tips["tip"] <= 9]
```

Comparison with spreadsheets

Since many potential pandas users have some familiarity with spreadsheet programs like Excel, this page is meant to provide some examples of how various spreadsheet operations would be performed using pandas. This page will use terminology and link to documentation for Excel, but much will be the same/similar in Google Sheets, LibreOffice Calc, Apple Numbers, and other Excel-compatible spreadsheet software.

If you're new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

```
In [1]: import pandas as pd

In [2]: import numpy as np
```

Data structures

General terminology translation

pandas	Excel
`DataFrame`	worksheet
`Series`	column
`Index`	row headings
row	row
`NaN`	empty cell

`DataFrame`

A `DataFrame` in pandas is analogous to an Excel worksheet. While an Excel workbook can contain multiple worksheets, pandas `DataFrame`s exist independently.

`Series`

A `Series` is the data structure that represents one column of a `DataFrame`. Working with a `Series` is analogous to referencing a column of a spreadsheet.

`Index`

Every `DataFrame` and `Series` has an `Index`, which are labels on the *rows* of the data. In pandas, if no index is specified, a *`RangeIndex`* is used by default (first row = 0, second row = 1, and so on), analogous to row headings/numbers in spreadsheets.

In pandas, indexes can be set to one (or multiple) unique values, which is like having a column that is used as the row identifier in a worksheet. Unlike most spreadsheets, these `Index` values can actually be used to reference the rows. (Note that this can be done in Excel with structured references.) For example, in spreadsheets, you would reference the first row as `A1:Z1`, while in pandas you could use `populations.loc['Chicago']`.

Index values are also persistent, so if you re-order the rows in a `DataFrame`, the label for a particular row don't change.

See the *indexing documentation* for much more on how to use an `Index` effectively.

Copies vs. in place operations

Most pandas operations return copies of the `Series`/`DataFrame`. To make the changes "stick", you'll need to either assign to a new variable:

```
sorted_df = df.sort_values("col1")
```

or overwrite the original one:

```
df = df.sort_values("col1")
```

Note: You will see an `inplace=True` keyword argument available for some methods:

```
df.sort_values("col1", inplace=True)
```

Its use is discouraged. *More information.*

Data input / output

Constructing a DataFrame from values

In a spreadsheet, values can be typed directly into cells.

A pandas DataFrame can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

```
In [3]: df = pd.DataFrame({"x": [1, 3, 5], "y": [2, 4, 6]})

In [4]: df
Out[4]:
   x  y
0  1  2
1  3  4
2  5  6
```

Reading external data

Both Excel and *pandas* can import data from various sources in various formats.

CSV

Let's load and display the tips dataset from the pandas tests, which is a CSV file. In Excel, you would download and then open the CSV. In pandas, you pass the URL or local path of the CSV file to *read_csv()*:

```
In [5]: url = (
   ...:     "https://raw.github.com/pandas-dev"
   ...:     "/pandas/main/pandas/tests/io/data/csv/tips.csv"
   ...: )
   ...:

In [6]: tips = pd.read_csv(url)

In [7]: tips
Out[7]:
     total_bill   tip     sex smoker   day    time  size
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
4         24.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       29.03  5.92    Male     No   Sat  Dinner     3
```

(continues on next page)

(continued from previous page)

```
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]
```

Like Excel's Text Import Wizard, `read_csv` can take a number of parameters to specify how the data should be parsed. For example, if the data was instead tab delimited, and did not have column names, the pandas command would be:

```
tips = pd.read_csv("tips.csv", sep="\t", header=None)

# alternatively, read_table is an alias to read_csv with tab delimiter
tips = pd.read_table("tips.csv", header=None)
```

Excel files

Excel opens various Excel file formats by double-clicking them, or using the Open menu. In pandas, you use *special methods for reading and writing from/to Excel files*.

Let's first *create a new Excel file* based on the `tips` dataframe in the above example:

```
tips.to_excel("./tips.xlsx")
```

Should you wish to subsequently access the data in the `tips.xlsx` file, you can read it into your module using

```
tips_df = pd.read_excel("./tips.xlsx", index_col=0)
```

You have just read in an Excel file using pandas!

Limiting output

Spreadsheet programs will only show one screenful of data at a time and then allow you to scroll, so there isn't really a need to limit output. In pandas, you'll need to put a little more thought into controlling how your `DataFrames` are displayed.

By default, pandas will truncate output of large `DataFrames` to show the first and last rows. This can be overridden by *changing the pandas options*, or using `DataFrame.head()` or `DataFrame.tail()`.

```
In [8]: tips.head(5)
Out[8]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

Exporting data

By default, desktop spreadsheet software will save to its respective file format (`.xlsx`, `.ods`, etc). You can, however, save to other file formats.

pandas can create Excel files, *CSV*, or *a number of other formats*.

Data operations

Operations on columns

In spreadsheets, formulas are often created in individual cells and then dragged into other cells to compute them for other columns. In pandas, you're able to do operations on whole columns directly.

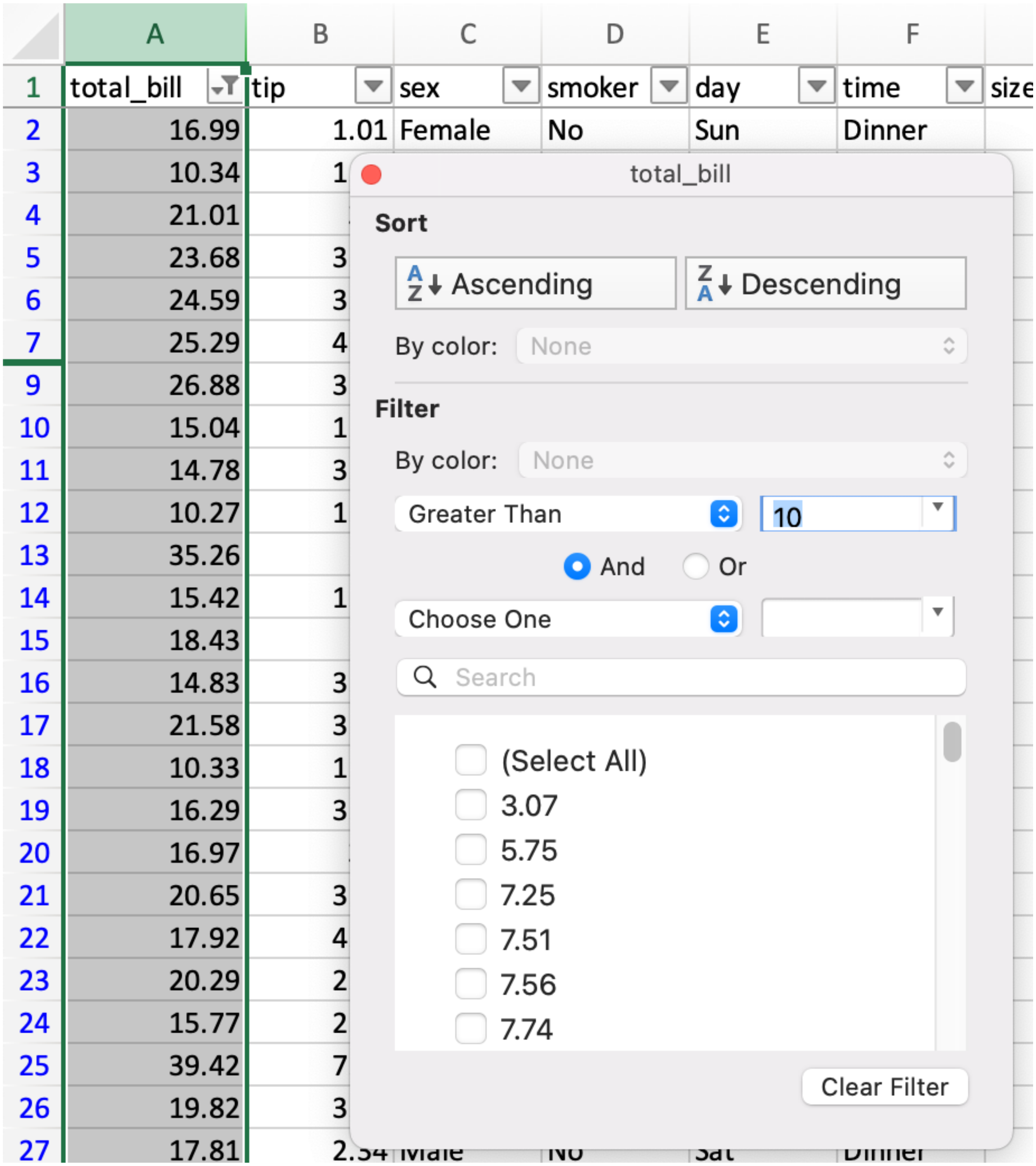
pandas provides vectorized operations by specifying the individual `Series` in the `DataFrame`. New columns can be assigned in the same way. The *`DataFrame.drop()`* method drops a column from the `DataFrame`.

```
In [9]: tips["total_bill"] = tips["total_bill"] - 2

In [10]: tips["new_bill"] = tips["total_bill"] / 2

In [11]: tips
Out[11]:
     total_bill   tip     sex smoker   day    time  size  new_bill
0         14.99  1.01  Female     No   Sun  Dinner     2     7.495
1          8.34  1.66    Male     No   Sun  Dinner     3     4.170
2         19.01  3.50    Male     No   Sun  Dinner     3     9.505
3         21.68  3.31    Male     No   Sun  Dinner     2    10.840
4         22.59  3.61  Female     No   Sun  Dinner     4    11.295
..          ...   ...     ...    ...   ...     ...   ...       ...
239       27.03  5.92    Male     No   Sat  Dinner     3    13.515
240       25.18  2.00  Female    Yes   Sat  Dinner     2    12.590
241       20.67  2.00    Male    Yes   Sat  Dinner     2    10.335
242       15.82  1.75    Male     No   Sat  Dinner     2     7.910
243       16.78  3.00  Female     No  Thur  Dinner     2     8.390

[244 rows x 8 columns]

In [12]: tips = tips.drop("new_bill", axis=1)
```

Note that we aren't having to tell it to do that subtraction cell-by-cell — pandas handles that for us. See *how to create new columns derived from existing columns*.

Filtering

In Excel, filtering is done through a graphical menu.



DataFrames can be filtered in multiple ways; the most intuitive of which is using *boolean indexing*.

```
In [13]: tips[tips["total_bill"] > 10]
Out[13]:
     total_bill   tip     sex smoker   day    time  size
```

(continues on next page)

(continued from previous page)

```
0          14.99  1.01  Female     No   Sun  Dinner     2
2          19.01  3.50    Male     No   Sun  Dinner     3
3          21.68  3.31    Male     No   Sun  Dinner     2
4          22.59  3.61  Female     No   Sun  Dinner     4
5          23.29  4.71    Male     No   Sun  Dinner     4
..           ...   ...     ...    ...   ...     ...   ...
239        27.03  5.92    Male     No   Sat  Dinner     3
240        25.18  2.00  Female    Yes   Sat  Dinner     2
241        20.67  2.00    Male    Yes   Sat  Dinner     2
242        15.82  1.75    Male     No   Sat  Dinner     2
243        16.78  3.00  Female     No  Thur  Dinner     2

[204 rows x 7 columns]
```

The above statement is simply passing a `Series` of `True/False` objects to the DataFrame, returning all rows with `True`.

```
In [14]: is_dinner = tips["time"] == "Dinner"

In [15]: is_dinner
Out[15]:
0      True
1      True
2      True
3      True
4      True
       ...
239    True
240    True
241    True
242    True
243    True
Name: time, Length: 244, dtype: bool

In [16]: is_dinner.value_counts()
Out[16]:
True     176
False     68
Name: time, dtype: int64

In [17]: tips[is_dinner]
Out[17]:
     total_bill   tip     sex smoker   day    time  size
0          14.99  1.01  Female     No   Sun  Dinner     2
1           8.34  1.66    Male     No   Sun  Dinner     3
2          19.01  3.50    Male     No   Sun  Dinner     3
3          21.68  3.31    Male     No   Sun  Dinner     2
4          22.59  3.61  Female     No   Sun  Dinner     4
..           ...   ...     ...    ...   ...     ...   ...
239        27.03  5.92    Male     No   Sat  Dinner     3
240        25.18  2.00  Female    Yes   Sat  Dinner     2
241        20.67  2.00    Male    Yes   Sat  Dinner     2
```

(continues on next page)

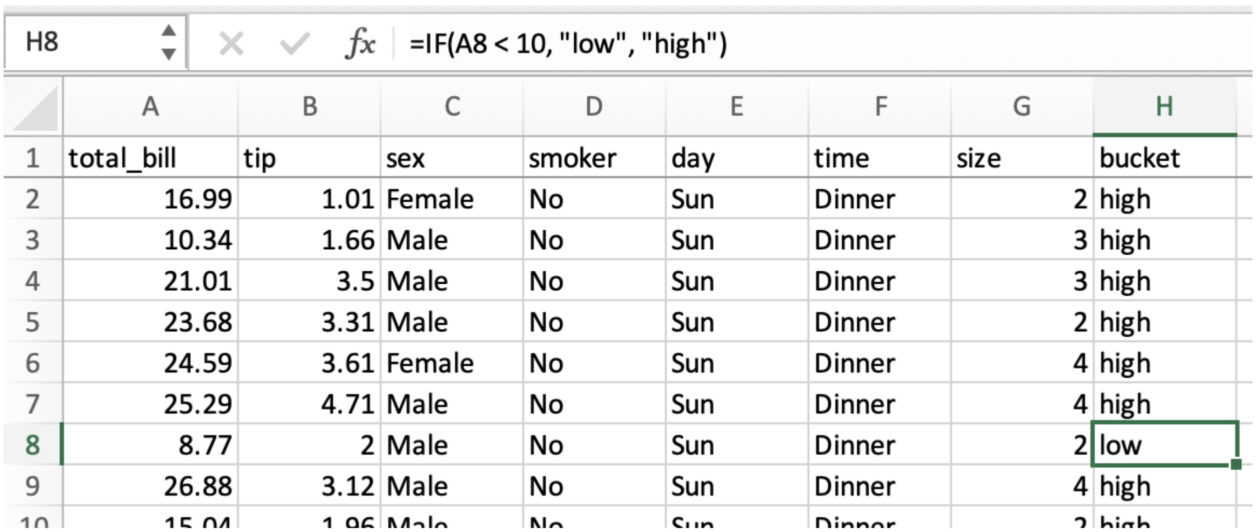
(continued from previous page)

```
242        15.82  1.75    Male     No   Sat  Dinner     2
243        16.78  3.00  Female     No  Thur  Dinner     2

[176 rows x 7 columns]
```

If/then logic

Let's say we want to make a `bucket` column with values of `low` and `high`, based on whether the `total_bill` is less or more than $10.

In spreadsheets, logical comparison can be done with conditional formulas. We'd use a formula of `=IF(A2 < 10, "low", "high")`, dragged to all cells in a new `bucket` column.



H8 =IF(A8 < 10, "low", "high")

	A	B	C	D	E	F	G	H
1	total_bill	tip	sex	smoker	day	time	size	bucket
2	16.99	1.01	Female	No	Sun	Dinner	2	high
3	10.34	1.66	Male	No	Sun	Dinner	3	high
4	21.01	3.5	Male	No	Sun	Dinner	3	high
5	23.68	3.31	Male	No	Sun	Dinner	2	high
6	24.59	3.61	Female	No	Sun	Dinner	4	high
7	25.29	4.71	Male	No	Sun	Dinner	4	high
8	8.77	2	Male	No	Sun	Dinner	2	low
9	26.88	3.12	Male	No	Sun	Dinner	4	high

The same operation in pandas can be accomplished using the `where` method from `numpy`.

```
In [18]: tips["bucket"] = np.where(tips["total_bill"] < 10, "low", "high")

In [19]: tips
Out[19]:
     total_bill   tip     sex smoker   day    time  size bucket
0         14.99  1.01  Female     No   Sun  Dinner     2   high
1          8.34  1.66    Male     No   Sun  Dinner     3    low
2         19.01  3.50    Male     No   Sun  Dinner     3   high
3         21.68  3.31    Male     No   Sun  Dinner     2   high
4         22.59  3.61  Female     No   Sun  Dinner     4   high
..          ...   ...     ...    ...   ...     ...   ...    ...
239       27.03  5.92    Male     No   Sat  Dinner     3   high
240       25.18  2.00  Female    Yes   Sat  Dinner     2   high
241       20.67  2.00    Male    Yes   Sat  Dinner     2   high
242       15.82  1.75    Male     No   Sat  Dinner     2   high
243       16.78  3.00  Female     No  Thur  Dinner     2   high

[244 rows x 8 columns]
```

Date functionality

This section will refer to "dates", but timestamps are handled similarly.

We can think of date functionality in two parts: parsing, and output. In spreadsheets, date values are generally parsed automatically, though there is a DATEVALUE function if you need it. In pandas, you need to explicitly convert plain text to datetime objects, either *while reading from a CSV* or *once in a DataFrame*.

Once parsed, spreadsheets display the dates in a default format, though the format can be changed. In pandas, you'll generally want to keep dates as `datetime` objects while you're doing calculations with them. Outputting *parts* of dates (such as the year) is done through date functions in spreadsheets, and *datetime properties* in pandas.

Given `date1` and `date2` in columns `A` and `B` of a spreadsheet, you might have these formulas:

column	formula
`date1_year`	`=YEAR(A2)`
`date2_month`	`=MONTH(B2)`
`date1_next`	`=DATE(YEAR(A2),MONTH(A2)+1,1)`
`months_between`	`=DATEDIF(A2,B2,"M")`

The equivalent pandas operations are shown below.

```
In [20]: tips["date1"] = pd.Timestamp("2013-01-15")

In [21]: tips["date2"] = pd.Timestamp("2015-02-15")

In [22]: tips["date1_year"] = tips["date1"].dt.year

In [23]: tips["date2_month"] = tips["date2"].dt.month

In [24]: tips["date1_next"] = tips["date1"] + pd.offsets.MonthBegin()

In [25]: tips["months_between"] = tips["date2"].dt.to_period("M") - tips[
   ....:     "date1"
   ....: ].dt.to_period("M")
   ....: 

In [26]: tips[
   ....:     ["date1", "date2", "date1_year", "date2_month", "date1_next", "months_
↪between"]
   ....: ]
   ....: 
Out[26]: 
          date1      date2  date1_year  date2_month date1_next    months_between
0    2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
1    2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
2    2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
3    2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
4    2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
..          ...        ...         ...          ...        ...               ...
239 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
240 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
241 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
242 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
```

(continues on next page)

(continued from previous page)

```
243 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>

[244 rows x 6 columns]
```

See *Time series / date functionality* for more details.

Selection of columns

In spreadsheets, you can select columns you want by:

- Hiding columns
- Deleting columns
- Referencing a range from one worksheet into another

Since spreadsheet columns are typically named in a header row, renaming a column is simply a matter of changing the text in that first cell.

The same operations are expressed in pandas below.

Keep certain columns

```
In [27]: tips[["sex", "total_bill", "tip"]]
Out[27]:
        sex  total_bill   tip
0    Female       14.99  1.01
1      Male        8.34  1.66
2      Male       19.01  3.50
3      Male       21.68  3.31
4    Female       22.59  3.61
..      ...         ...   ...
239    Male       27.03  5.92
240  Female       25.18  2.00
241    Male       20.67  2.00
242    Male       15.82  1.75
243  Female       16.78  3.00

[244 rows x 3 columns]
```

Drop a column

```
In [28]: tips.drop("sex", axis=1)
Out[28]:
     total_bill   tip smoker   day    time  size
0         14.99  1.01     No   Sun  Dinner     2
1          8.34  1.66     No   Sun  Dinner     3
2         19.01  3.50     No   Sun  Dinner     3
3         21.68  3.31     No   Sun  Dinner     2
4         22.59  3.61     No   Sun  Dinner     4
```

(continues on next page)

(continued from previous page)

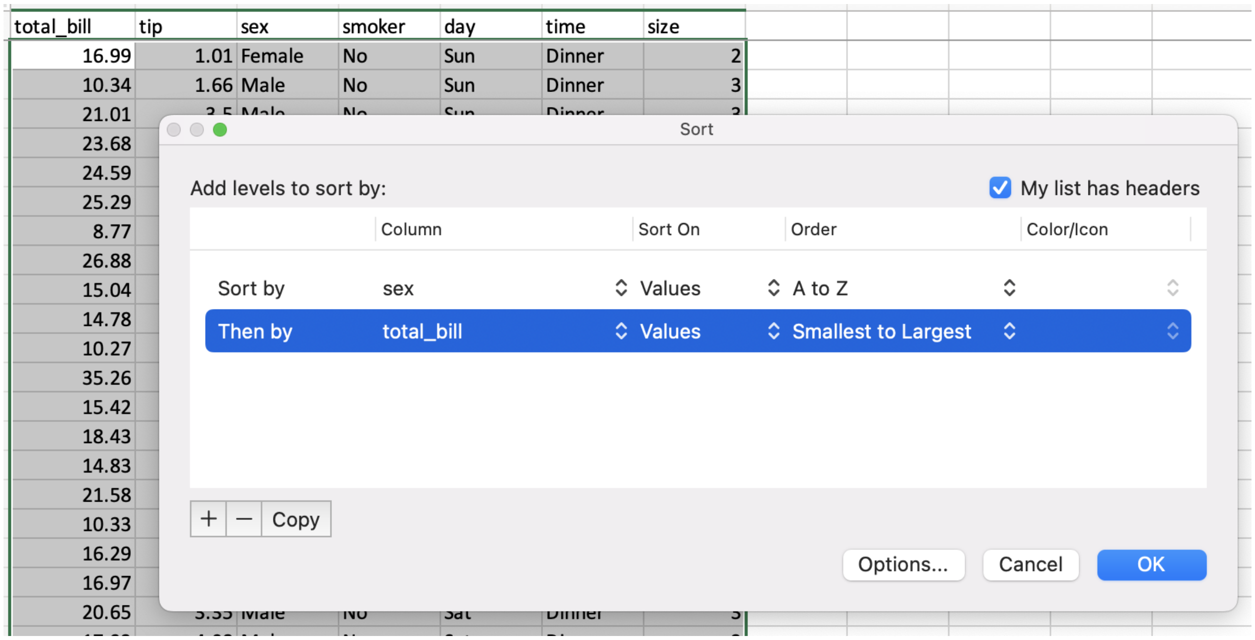
```
..          ...   ...    ...   ...     ...   ...
239       27.03  5.92     No   Sat  Dinner     3
240       25.18  2.00    Yes   Sat  Dinner     2
241       20.67  2.00    Yes   Sat  Dinner     2
242       15.82  1.75     No   Sat  Dinner     2
243       16.78  3.00     No  Thur  Dinner     2

[244 rows x 6 columns]
```

Rename a column

```
In [29]: tips.rename(columns={"total_bill": "total_bill_2"})
Out[29]:
     total_bill_2   tip     sex smoker   day    time  size
0           14.99  1.01  Female     No   Sun  Dinner     2
1            8.34  1.66    Male     No   Sun  Dinner     3
2           19.01  3.50    Male     No   Sun  Dinner     3
3           21.68  3.31    Male     No   Sun  Dinner     2
4           22.59  3.61  Female     No   Sun  Dinner     4
..            ...   ...     ...    ...   ...     ...   ...
239         27.03  5.92    Male     No   Sat  Dinner     3
240         25.18  2.00  Female    Yes   Sat  Dinner     2
241         20.67  2.00    Male    Yes   Sat  Dinner     2
242         15.82  1.75    Male     No   Sat  Dinner     2
243         16.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]
```

Sorting by values

Sorting in spreadsheets is accomplished via the sort dialog.



pandas has a *DataFrame.sort_values()* method, which takes a list of columns to sort by.

```
In [30]: tips = tips.sort_values(["sex", "total_bill"])

In [31]: tips
Out[31]:
     total_bill    tip     sex smoker   day    time  size
67         1.07   1.00  Female    Yes   Sat  Dinner     1
92         3.75   1.00  Female    Yes   Fri  Dinner     2
111        5.25   1.00  Female     No   Sat  Dinner     1
145        6.35   1.50  Female     No  Thur   Lunch     2
135        6.51   1.25  Female     No  Thur   Lunch     2
..          ...    ...     ...    ...   ...     ...   ...
182       43.35   3.50    Male    Yes   Sun  Dinner     3
156       46.17   5.00    Male     No   Sun  Dinner     6
59        46.27   6.73    Male     No   Sat  Dinner     4
212       46.33   9.00    Male     No   Sat  Dinner     4
170       48.81  10.00    Male    Yes   Sat  Dinner     3

[244 rows x 7 columns]
```

String processing

Finding length of string

In spreadsheets, the number of characters in text can be found with the LEN function. This can be used with the TRIM function to remove extra whitespace.

```
=LEN(TRIM(A2))
```

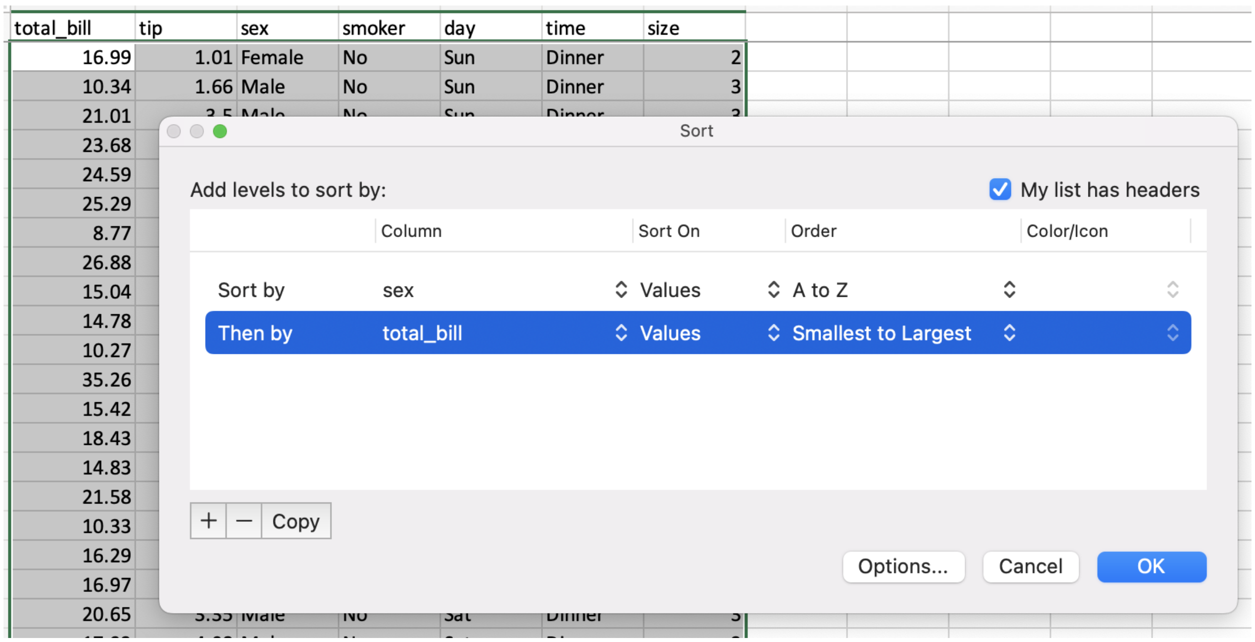
You can find the length of a character string with *Series.str.len()*. In Python 3, all strings are Unicode strings. `len` includes trailing blanks. Use `len` and `rstrip` to exclude trailing blanks.

```
In [32]: tips["time"].str.len()
Out[32]:
67     6
92     6
111    6
145    5
135    5
      ..
182    6
156    6
59     6
212    6
170    6
Name: time, Length: 244, dtype: int64

In [33]: tips["time"].str.rstrip().str.len()
Out[33]:
67     6
92     6
111    6
145    5
135    5
      ..
182    6
156    6
59     6
212    6
170    6
Name: time, Length: 244, dtype: int64
```

Note this will still include multiple spaces within the string, so isn't 100% equivalent.

Finding position of substring

The FIND spreadsheet function returns the position of a substring, with the first character being 1.



You can find the position of a character in a column of strings with the `Series.str.find()` method. `find` searches for the first position of the substring. If the substring is found, the method returns its position. If not found, it returns `-1`. Keep in mind that Python indexes are zero-based.

```
In [34]: tips["sex"].str.find("ale")
Out[34]:
67     3
92     3
111    3
145    3
135    3
      ..
182    1
156    1
59     1
212    1
170    1
Name: sex, Length: 244, dtype: int64
```

Extracting substring by position

Spreadsheets have a MID formula for extracting a substring from a given position. To get the first character:

```
=MID(A2,1,1)
```

With pandas you can use `[]` notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [35]: tips["sex"].str[0:1]
Out[35]:
67     F
92     F
```

(continues on next page)

(continued from previous page)

```
111    F
145    F
135    F
      ..
182    M
156    M
59     M
212    M
170    M
Name: sex, Length: 244, dtype: object
```

Extracting nth word

In Excel, you might use the Text to Columns Wizard for splitting text and retrieving a specific column. (Note it's possible to do so through a formula as well.)

The simplest way to extract words in pandas is to split the strings by spaces, then reference the word by index. Note there are more powerful approaches should you need them.

```
In [36]: firstlast = pd.DataFrame({"String": ["John Smith", "Jane Cook"]})

In [37]: firstlast["First_Name"] = firstlast["String"].str.split(" ", expand=True)[0]

In [38]: firstlast["Last_Name"] = firstlast["String"].str.rsplit(" ", expand=True)[1]

In [39]: firstlast
Out[39]:
       String First_Name Last_Name
0  John Smith       John     Smith
1   Jane Cook       Jane      Cook
```

Changing case

Spreadsheets provide UPPER, LOWER, and PROPER functions for converting text to upper, lower, and title case, respectively.

The equivalent pandas methods are `Series.str.upper()`, `Series.str.lower()`, and `Series.str.title()`.

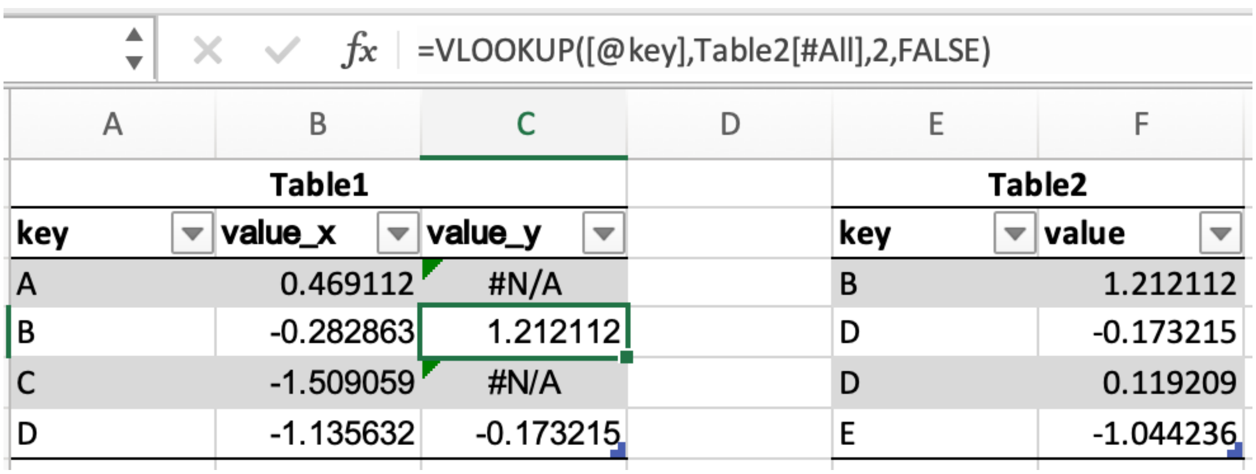
```
In [40]: firstlast = pd.DataFrame({"string": ["John Smith", "Jane Cook"]})

In [41]: firstlast["upper"] = firstlast["string"].str.upper()

In [42]: firstlast["lower"] = firstlast["string"].str.lower()

In [43]: firstlast["title"] = firstlast["string"].str.title()

In [44]: firstlast
Out[44]:
       string       upper       lower       title
0  John Smith  JOHN SMITH  john smith  John Smith
1   Jane Cook   JANE COOK   jane cook   Jane Cook
```

Merging

The following tables will be used in the merge examples:

```
In [45]: df1 = pd.DataFrame({"key": ["A", "B", "C", "D"], "value": np.random.randn(4)})

In [46]: df1
Out[46]:
  key     value
0   A  0.469112
1   B -0.282863
2   C -1.509059
3   D -1.135632

In [47]: df2 = pd.DataFrame({"key": ["B", "D", "D", "E"], "value": np.random.randn(4)})

In [48]: df2
Out[48]:
  key     value
0   B  1.212112
1   D -0.173215
2   D  0.119209
3   E -1.044236
```

In Excel, there are merging of tables can be done through a VLOOKUP.



key	value_x	value_y		key	value
A	0.469112	#N/A		B	1.212112
B	-0.282863	1.212112		D	-0.173215
C	-1.509059	#N/A		D	0.119209
D	-1.135632	-0.173215		E	-1.044236

pandas DataFrames have a *merge()* method, which provides similar functionality. The data does not have to be sorted ahead of time, and different join types are accomplished via the how keyword.

```
In [49]: inner_join = df1.merge(df2, on=["key"], how="inner")

In [50]: inner_join
Out[50]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209
```

(continues on next page)

(continued from previous page)

```
In [51]: left_join = df1.merge(df2, on=["key"], how="left")

In [52]: left_join
Out[52]: 
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209

In [53]: right_join = df1.merge(df2, on=["key"], how="right")

In [54]: right_join
Out[54]: 
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209
3   E       NaN -1.044236

In [55]: outer_join = df1.merge(df2, on=["key"], how="outer")

In [56]: outer_join
Out[56]: 
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236
```

`merge` has a number of advantages over `VLOOKUP`:

- The lookup value doesn't need to be the first column of the lookup table
- If multiple rows are matched, there will be one row for each match, instead of just the first
- It will include all columns from the lookup table, instead of just a single specified column
- It supports *more complex join operations*

Other considerations

Fill Handle

Create a series of numbers following a set pattern in a certain set of cells. In a spreadsheet, this would be done by shift+drag after entering the first number or by entering the first two or three values and then dragging.

This can be achieved by creating a series and assigning it to the desired cells.

```
In [57]: df = pd.DataFrame({"AAA": [1] * 8, "BBB": list(range(0, 8))})

In [58]: df
Out[58]:
   AAA  BBB
0    1    0
1    1    1
2    1    2
3    1    3
4    1    4
5    1    5
6    1    6
7    1    7

In [59]: series = list(range(1, 5))

In [60]: series
Out[60]: [1, 2, 3, 4]

In [61]: df.loc[2:5, "AAA"] = series

In [62]: df
Out[62]:
   AAA  BBB
0    1    0
1    1    1
2    1    2
3    2    3
4    3    4
5    4    5
6    1    6
7    1    7
```

Drop Duplicates

Excel has built-in functionality for removing duplicate values. This is supported in pandas via *drop_duplicates()*.

```
In [63]: df = pd.DataFrame(
   ....:     {
   ....:         "class": ["A", "A", "A", "B", "C", "D"],
   ....:         "student_count": [42, 35, 42, 50, 47, 45],
   ....:         "all_pass": ["Yes", "Yes", "Yes", "No", "No", "Yes"],
   ....:     }
   ....: )
   ....:

In [64]: df.drop_duplicates()
Out[64]:
  class  student_count all_pass
0     A             42      Yes
1     A             35      Yes
```

(continues on next page)

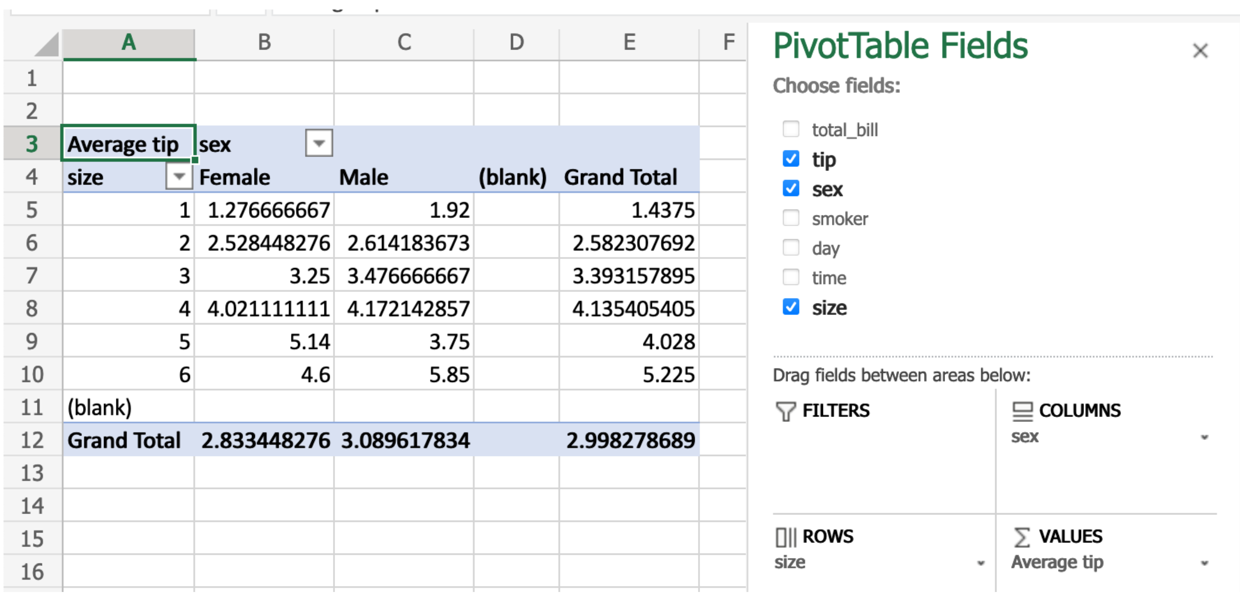
(continued from previous page)

```
3      B             50        No
4      C             47        No
5      D             45       Yes

In [65]: df.drop_duplicates(["class", "student_count"])
Out[65]:
  class  student_count all_pass
0      A             42      Yes
1      A             35      Yes
3      B             50       No
4      C             47       No
5      D             45      Yes
```

Pivot Tables

PivotTables from spreadsheets can be replicated in pandas through *Reshaping and pivot tables*. Using the `tips` dataset again, let's find the average gratuity by size of the party and sex of the server.

In Excel, we use the following configuration for the PivotTable:



The equivalent in pandas:

```
In [66]: pd.pivot_table(
   ....:     tips, values="tip", index=["size"], columns=["sex"], aggfunc=np.average
   ....: )
   ....:
Out[66]:
sex      Female      Male
size
1      1.276667  1.920000
2      2.528448  2.614184
3      3.250000  3.476667
4      4.021111  4.172143
```

(continues on next page)

(continued from previous page)

```
5     5.140000  3.750000
6     4.600000  5.850000
```

Adding a row

Assuming we are using a *RangeIndex* (numbered `0`, `1`, etc.), we can use *concat()* to add a row to the bottom of a `DataFrame`.

```
In [67]: df
Out[67]:
  class  student_count all_pass
0     A             42      Yes
1     A             35      Yes
2     A             42      Yes
3     B             50       No
4     C             47       No
5     D             45      Yes

In [68]: new_row = pd.DataFrame([["E", 51, True]],
   ....:                        columns=["class", "student_count", "all_pass"])
   ....:

In [69]: pd.concat([df, new_row])
Out[69]:
  class  student_count all_pass
0     A             42      Yes
1     A             35      Yes
2     A             42      Yes
3     B             50       No
4     C             47       No
5     D             45      Yes
0     E             51     True
```

Find and Replace

Excel's Find dialog takes you to cells that match, one by one. In pandas, this operation is generally done for an entire column or `DataFrame` at once through *conditional expressions*.

```
In [70]: tips
Out[70]:
     total_bill   tip     sex smoker   day    time  size
67         1.07  1.00  Female    Yes   Sat  Dinner     1
92         3.75  1.00  Female    Yes   Fri  Dinner     2
111        5.25  1.00  Female     No   Sat  Dinner     1
145        6.35  1.50  Female     No  Thur   Lunch     2
135        6.51  1.25  Female     No  Thur   Lunch     2
..          ...   ...     ...    ...   ...     ...   ...
182       43.35  3.50    Male    Yes   Sun  Dinner     3
156       46.17  5.00    Male     No   Sun  Dinner     6
```

(continues on next page)

(continued from previous page)

```
59         46.27   6.73    Male     No  Sat  Dinner     4
212        46.33   9.00    Male     No  Sat  Dinner     4
170        48.81  10.00    Male    Yes  Sat  Dinner     3

[244 rows x 7 columns]

In [71]: tips == "Sun"
Out[71]:
     total_bill    tip    sex  smoker    day   time   size
67        False  False  False   False  False  False  False
92        False  False  False   False  False  False  False
111       False  False  False   False  False  False  False
145       False  False  False   False  False  False  False
135       False  False  False   False  False  False  False
..          ...    ...    ...     ...    ...    ...    ...
182       False  False  False   False   True  False  False
156       False  False  False   False   True  False  False
59        False  False  False   False  False  False  False
212       False  False  False   False  False  False  False
170       False  False  False   False  False  False  False

[244 rows x 7 columns]

In [72]: tips["day"].str.contains("S")
Out[72]:
67      True
92     False
111     True
145    False
135    False
       ...
182     True
156     True
59      True
212     True
170     True
Name: day, Length: 244, dtype: bool
```

pandas' *replace()* is comparable to Excel's `Replace All`.

```
In [73]: tips.replace("Thu", "Thursday")
Out[73]:
     total_bill   tip     sex smoker   day    time  size
67         1.07  1.00  Female    Yes   Sat  Dinner     1
92         3.75  1.00  Female    Yes   Fri  Dinner     2
111        5.25  1.00  Female     No   Sat  Dinner     1
145        6.35  1.50  Female     No  Thur   Lunch     2
135        6.51  1.25  Female     No  Thur   Lunch     2
..          ...   ...     ...    ...   ...     ...   ...
182       43.35  3.50    Male    Yes   Sun  Dinner     3
156       46.17  5.00    Male     No   Sun  Dinner     6
59        46.27  6.73    Male     No   Sat  Dinner     4
```

(continues on next page)

(continued from previous page)

```
212       46.33   9.00    Male     No   Sat  Dinner     4
170       48.81  10.00    Male    Yes   Sat  Dinner     3

[244 rows x 7 columns]
```

Comparison with SAS

For potential users coming from SAS this page is meant to demonstrate how different SAS operations would be performed in pandas.

If you're new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

```
In [1]: import pandas as pd

In [2]: import numpy as np
```

Data structures

General terminology translation

pandas	SAS
`DataFrame`	data set
column	variable
row	observation
groupby	BY-group
`NaN`	`.`

`DataFrame`

A `DataFrame` in pandas is analogous to a SAS data set - a two-dimensional data source with labeled columns that can be of different types. As will be shown in this document, almost any operation that can be applied to a data set using SAS's `DATA` step, can also be accomplished in pandas.

`Series`

A `Series` is the data structure that represents one column of a `DataFrame`. SAS doesn't have a separate data structure for a single column, but in general, working with a `Series` is analogous to referencing a column in the `DATA` step.

Index

Every DataFrame and Series has an Index - which are labels on the *rows* of the data. SAS does not have an exactly analogous concept. A data set's rows are essentially unlabeled, other than an implicit integer index that can be accessed during the DATA step (_N_).

In pandas, if no index is specified, an integer index is also used by default (first row = 0, second row = 1, and so on). While using a labeled Index or MultiIndex can enable sophisticated analyses and is ultimately an important part of pandas to understand, for this comparison we will essentially ignore the Index and just treat the DataFrame as a collection of columns. Please see the *indexing documentation* for much more on how to use an Index effectively.

Copies vs. in place operations

Most pandas operations return copies of the Series/DataFrame. To make the changes "stick", you'll need to either assign to a new variable:

```
sorted_df = df.sort_values("col1")
```

or overwrite the original one:

```
df = df.sort_values("col1")
```

Note: You will see an inplace=True keyword argument available for some methods:

```
df.sort_values("col1", inplace=True)
```

Its use is discouraged. *More information.*

Data input / output

Constructing a DataFrame from values

A SAS data set can be built from specified values by placing the data after a datalines statement and specifying the column names.

```
data df;
    input x y;
    datalines;
    1 2
    3 4
    5 6
    ;
run;
```

A pandas DataFrame can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

```
In [1]: df = pd.DataFrame({"x": [1, 3, 5], "y": [2, 4, 6]})

In [2]: df
```

(continues on next page)

(continued from previous page)

```
Out[2]:
   x  y
0  1  2
1  3  4
2  5  6
```

Reading external data

Like SAS, pandas provides utilities for reading in data from many formats. The `tips` dataset, found within the pandas tests (csv) will be used in many of the following examples.

SAS provides `PROC IMPORT` to read csv data into a data set.

```
proc import datafile='tips.csv' dbms=csv out=tips replace;
    getnames=yes;
run;
```

The pandas method is *read_csv()*, which works similarly.

```
In [3]: url = (
   ...:     "https://raw.github.com/pandas-dev/"
   ...:     "pandas/main/pandas/tests/io/data/csv/tips.csv"
   ...: )
   ...:

In [4]: tips = pd.read_csv(url)

In [5]: tips
Out[5]:
     total_bill   tip     sex smoker   day    time  size
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
4         24.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       29.03  5.92    Male     No   Sat  Dinner     3
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]
```

Like `PROC IMPORT`, `read_csv` can take a number of parameters to specify how the data should be parsed. For example, if the data was instead tab delimited, and did not have column names, the pandas command would be:

```
tips = pd.read_csv("tips.csv", sep="\t", header=None)

# alternatively, read_table is an alias to read_csv with tab delimiter
tips = pd.read_table("tips.csv", header=None)
```

In addition to text/csv, pandas supports a variety of other data formats such as Excel, HDF5, and SQL databases. These are all read via a `pd.read_*` function. See the *IO documentation* for more details.

Limiting output

By default, pandas will truncate output of large `DataFrames` to show the first and last rows. This can be overridden by *changing the pandas options*, or using `DataFrame.head()` or `DataFrame.tail()`.

```
In [1]: tips.head(5)
Out[1]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

The equivalent in SAS would be:

```
proc print data=df(obs=5);
run;
```

Exporting data

The inverse of `PROC IMPORT` in SAS is `PROC EXPORT`

```
proc export data=tips outfile='tips2.csv' dbms=csv;
run;
```

Similarly in pandas, the opposite of `read_csv` is `to_csv()`, and other data formats follow a similar api.

```
tips.to_csv("tips2.csv")
```

Data operations

Operations on columns

In the `DATA` step, arbitrary math expressions can be used on new or existing columns.

```
data tips;
    set tips;
    total_bill = total_bill - 2;
    new_bill = total_bill / 2;
run;
```

pandas provides vectorized operations by specifying the individual `Series` in the `DataFrame`. New columns can be assigned in the same way. The `DataFrame.drop()` method drops a column from the `DataFrame`.

```
In [1]: tips["total_bill"] = tips["total_bill"] - 2

In [2]: tips["new_bill"] = tips["total_bill"] / 2

In [3]: tips
Out[3]:
     total_bill   tip     sex smoker   day    time  size  new_bill
0         14.99  1.01  Female     No   Sun  Dinner     2     7.495
1          8.34  1.66    Male     No   Sun  Dinner     3     4.170
2         19.01  3.50    Male     No   Sun  Dinner     3     9.505
3         21.68  3.31    Male     No   Sun  Dinner     2    10.840
4         22.59  3.61  Female     No   Sun  Dinner     4    11.295
..          ...   ...     ...    ...   ...     ...   ...       ...
239       27.03  5.92    Male     No   Sat  Dinner     3    13.515
240       25.18  2.00  Female    Yes   Sat  Dinner     2    12.590
241       20.67  2.00    Male    Yes   Sat  Dinner     2    10.335
242       15.82  1.75    Male     No   Sat  Dinner     2     7.910
243       16.78  3.00  Female     No  Thur  Dinner     2     8.390

[244 rows x 8 columns]

In [4]: tips = tips.drop("new_bill", axis=1)
```

Filtering

Filtering in SAS is done with an `if` or `where` statement, on one or more columns.

```
data tips;
    set tips;
    if total_bill > 10;
run;

data tips;
    set tips;
    where total_bill > 10;
    /* equivalent in this case - where happens before the
       DATA step begins and can also be used in PROC statements */
run;
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using *boolean indexing*.

```
In [1]: tips[tips["total_bill"] > 10]
Out[1]:
     total_bill   tip     sex smoker   day    time  size
0         14.99  1.01  Female     No   Sun  Dinner     2
2         19.01  3.50    Male     No   Sun  Dinner     3
3         21.68  3.31    Male     No   Sun  Dinner     2
4         22.59  3.61  Female     No   Sun  Dinner     4
5         23.29  4.71    Male     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       27.03  5.92    Male     No   Sat  Dinner     3
240       25.18  2.00  Female    Yes   Sat  Dinner     2
```

(continues on next page)

(continued from previous page)

```
241       20.67  2.00    Male    Yes   Sat  Dinner     2
242       15.82  1.75    Male     No   Sat  Dinner     2
243       16.78  3.00  Female     No  Thur  Dinner     2

[204 rows x 7 columns]
```

The above statement is simply passing a `Series` of `True/False` objects to the DataFrame, returning all rows with `True`.

```
In [1]: is_dinner = tips["time"] == "Dinner"

In [2]: is_dinner
Out[2]:
0      True
1      True
2      True
3      True
4      True
       ...
239    True
240    True
241    True
242    True
243    True
Name: time, Length: 244, dtype: bool

In [3]: is_dinner.value_counts()
Out[3]:
True     176
False     68
Name: time, dtype: int64

In [4]: tips[is_dinner]
Out[4]:
     total_bill   tip     sex smoker   day    time  size
0         14.99  1.01  Female     No   Sun  Dinner     2
1          8.34  1.66    Male     No   Sun  Dinner     3
2         19.01  3.50    Male     No   Sun  Dinner     3
3         21.68  3.31    Male     No   Sun  Dinner     2
4         22.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       27.03  5.92    Male     No   Sat  Dinner     3
240       25.18  2.00  Female    Yes   Sat  Dinner     2
241       20.67  2.00    Male    Yes   Sat  Dinner     2
242       15.82  1.75    Male     No   Sat  Dinner     2
243       16.78  3.00  Female     No  Thur  Dinner     2

[176 rows x 7 columns]
```

If/then logic

In SAS, if/then logic can be used to create new columns.

```
data tips;
    set tips;
    format bucket $4.;

    if total_bill < 10 then bucket = 'low';
    else bucket = 'high';
run;
```

The same operation in pandas can be accomplished using the `where` method from `numpy`.

```
In [1]: tips["bucket"] = np.where(tips["total_bill"] < 10, "low", "high")

In [2]: tips
Out[2]: 
     total_bill   tip     sex smoker   day    time  size bucket
0         14.99  1.01  Female     No   Sun  Dinner     2   high
1          8.34  1.66    Male     No   Sun  Dinner     3    low
2         19.01  3.50    Male     No   Sun  Dinner     3   high
3         21.68  3.31    Male     No   Sun  Dinner     2   high
4         22.59  3.61  Female     No   Sun  Dinner     4   high
..          ...   ...     ...    ...   ...     ...   ...    ...
239       27.03  5.92    Male     No   Sat  Dinner     3   high
240       25.18  2.00  Female    Yes   Sat  Dinner     2   high
241       20.67  2.00    Male    Yes   Sat  Dinner     2   high
242       15.82  1.75    Male     No   Sat  Dinner     2   high
243       16.78  3.00  Female     No  Thur  Dinner     2   high

[244 rows x 8 columns]
```

Date functionality

SAS provides a variety of functions to do operations on date/datetime columns.

```
data tips;
    set tips;
    format date1 date2 date1_plusmonth mmddyy10.;
    date1 = mdy(1, 15, 2013);
    date2 = mdy(2, 15, 2015);
    date1_year = year(date1);
    date2_month = month(date2);
    * shift date to beginning of next interval;
    date1_next = intnx('MONTH', date1, 1);
    * count intervals between dates;
    months_between = intck('MONTH', date1, date2);
run;
```

The equivalent pandas operations are shown below. In addition to these functions pandas supports other Time Series features not available in Base SAS (such as resampling and custom offsets) - see the *timeseries documentation* for more details.

```
In [1]: tips["date1"] = pd.Timestamp("2013-01-15")

In [2]: tips["date2"] = pd.Timestamp("2015-02-15")

In [3]: tips["date1_year"] = tips["date1"].dt.year

In [4]: tips["date2_month"] = tips["date2"].dt.month

In [5]: tips["date1_next"] = tips["date1"] + pd.offsets.MonthBegin()

In [6]: tips["months_between"] = tips["date2"].dt.to_period("M") - tips[
   ...:     "date1"
   ...: ].dt.to_period("M")
   ...:

In [7]: tips[
   ...:     ["date1", "date2", "date1_year", "date2_month", "date1_next", "months_between
↪"]
   ...: ]
   ...:
Out[7]:
         date1      date2  date1_year  date2_month date1_next    months_between
0   2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
1   2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
2   2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
3   2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
4   2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
..         ...        ...         ...          ...        ...               ...
239 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
240 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
241 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
242 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
243 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>

[244 rows x 6 columns]
```

Selection of columns

SAS provides keywords in the DATA step to select, drop, and rename columns.

```
data tips;
    set tips;
    keep sex total_bill tip;
run;

data tips;
    set tips;
    drop sex;
run;

data tips;
```

(continues on next page)

(continued from previous page)

```
    set tips;
    rename total_bill=total_bill_2;
run;
```

The same operations are expressed in pandas below.

Keep certain columns

```
In [1]: tips[["sex", "total_bill", "tip"]]
Out[1]:
        sex  total_bill   tip
0    Female       14.99  1.01
1      Male        8.34  1.66
2      Male       19.01  3.50
3      Male       21.68  3.31
4    Female       22.59  3.61
..      ...         ...   ...
239    Male       27.03  5.92
240  Female       25.18  2.00
241    Male       20.67  2.00
242    Male       15.82  1.75
243  Female       16.78  3.00

[244 rows x 3 columns]
```

Drop a column

```
In [2]: tips.drop("sex", axis=1)
Out[2]:
     total_bill   tip smoker   day    time  size
0         14.99  1.01     No   Sun  Dinner     2
1          8.34  1.66     No   Sun  Dinner     3
2         19.01  3.50     No   Sun  Dinner     3
3         21.68  3.31     No   Sun  Dinner     2
4         22.59  3.61     No   Sun  Dinner     4
..          ...   ...    ...   ...     ...   ...
239       27.03  5.92     No   Sat  Dinner     3
240       25.18  2.00    Yes   Sat  Dinner     2
241       20.67  2.00    Yes   Sat  Dinner     2
242       15.82  1.75     No   Sat  Dinner     2
243       16.78  3.00     No  Thur  Dinner     2

[244 rows x 6 columns]
```

Rename a column

```
In [1]: tips.rename(columns={"total_bill": "total_bill_2"})
Out[1]:
     total_bill_2   tip     sex smoker   day    time  size
0           14.99  1.01  Female     No   Sun  Dinner     2
1            8.34  1.66    Male     No   Sun  Dinner     3
2           19.01  3.50    Male     No   Sun  Dinner     3
3           21.68  3.31    Male     No   Sun  Dinner     2
4           22.59  3.61  Female     No   Sun  Dinner     4
..            ...   ...     ...    ...   ...     ...   ...
239         27.03  5.92    Male     No   Sat  Dinner     3
240         25.18  2.00  Female    Yes   Sat  Dinner     2
241         20.67  2.00    Male    Yes   Sat  Dinner     2
242         15.82  1.75    Male     No   Sat  Dinner     2
243         16.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]
```

Sorting by values

Sorting in SAS is accomplished via `PROC SORT`

```
proc sort data=tips;
    by sex total_bill;
run;
```

pandas has a *DataFrame.sort_values()* method, which takes a list of columns to sort by.

```
In [1]: tips = tips.sort_values(["sex", "total_bill"])

In [2]: tips
Out[2]:
     total_bill    tip     sex smoker   day    time  size
67         1.07   1.00  Female    Yes   Sat  Dinner     1
92         3.75   1.00  Female    Yes   Fri  Dinner     2
111        5.25   1.00  Female     No   Sat  Dinner     1
145        6.35   1.50  Female     No  Thur   Lunch     2
135        6.51   1.25  Female     No  Thur   Lunch     2
..          ...    ...     ...    ...   ...     ...   ...
182       43.35   3.50    Male    Yes   Sun  Dinner     3
156       46.17   5.00    Male     No   Sun  Dinner     6
59        46.27   6.73    Male     No   Sat  Dinner     4
212       46.33   9.00    Male     No   Sat  Dinner     4
170       48.81  10.00    Male    Yes   Sat  Dinner     3

[244 rows x 7 columns]
```

String processing

Finding length of string

SAS determines the length of a character string with the LENGTHN and LENGTHC functions. LENGTHN excludes trailing blanks and LENGTHC includes trailing blanks.

```
data _null_;
set tips;
put(LENGTHN(time));
put(LENGTHC(time));
run;
```

You can find the length of a character string with *Series.str.len()*. In Python 3, all strings are Unicode strings. len includes trailing blanks. Use len and rstrip to exclude trailing blanks.

```
In [1]: tips["time"].str.len()
Out[1]:
67     6
92     6
111    6
145    5
135    5
      ..
182    6
156    6
59     6
212    6
170    6
Name: time, Length: 244, dtype: int64

In [2]: tips["time"].str.rstrip().str.len()
Out[2]:
67     6
92     6
111    6
145    5
135    5
      ..
182    6
156    6
59     6
212    6
170    6
Name: time, Length: 244, dtype: int64
```

Finding position of substring

SAS determines the position of a character in a string with the FINDW function. `FINDW` takes the string defined by the first argument and searches for the first position of the substring you supply as the second argument.

```
data _null_;
set tips;
put(FINDW(sex,'ale'));
run;
```

You can find the position of a character in a column of strings with the `Series.str.find()` method. `find` searches for the first position of the substring. If the substring is found, the method returns its position. If not found, it returns `-1`. Keep in mind that Python indexes are zero-based.

```
In [1]: tips["sex"].str.find("ale")
Out[1]:
67     3
92     3
111    3
145    3
135    3
      ..
182    1
156    1
59     1
212    1
170    1
Name: sex, Length: 244, dtype: int64
```

Extracting substring by position

SAS extracts a substring from a string based on its position with the SUBSTR function.

```
data _null_;
set tips;
put(substr(sex,1,1));
run;
```

With pandas you can use `[]` notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [1]: tips["sex"].str[0:1]
Out[1]:
67     F
92     F
111    F
145    F
135    F
      ..
182    M
156    M
59     M
```

(continues on next page)

(continued from previous page)

```
212    M
170    M
Name: sex, Length: 244, dtype: object
```

Extracting nth word

The SAS SCAN function returns the nth word from a string. The first argument is the string you want to parse and the second argument specifies which word you want to extract.

```
data firstlast;
input String $60.;
First_Name = scan(string, 1);
Last_Name = scan(string, -1);
datalines2;
John Smith;
Jane Cook;
;;;
run;
```

The simplest way to extract words in pandas is to split the strings by spaces, then reference the word by index. Note there are more powerful approaches should you need them.

```
In [1]: firstlast = pd.DataFrame({"String": ["John Smith", "Jane Cook"]})

In [2]: firstlast["First_Name"] = firstlast["String"].str.split(" ", expand=True)[0]

In [3]: firstlast["Last_Name"] = firstlast["String"].str.rsplit(" ", expand=True)[1]

In [4]: firstlast
Out[4]:
       String First_Name Last_Name
0  John Smith       John     Smith
1   Jane Cook       Jane      Cook
```

Changing case

The SAS UPCASE LOWCASE and PROPCASE functions change the case of the argument.

```
data firstlast;
input String $60.;
string_up = UPCASE(string);
string_low = LOWCASE(string);
string_prop = PROPCASE(string);
datalines2;
John Smith;
Jane Cook;
;;;
run;
```

The equivalent pandas methods are `Series.str.upper()`, `Series.str.lower()`, and `Series.str.title()`.

```
In [1]: firstlast = pd.DataFrame({"string": ["John Smith", "Jane Cook"]})

In [2]: firstlast["upper"] = firstlast["string"].str.upper()

In [3]: firstlast["lower"] = firstlast["string"].str.lower()

In [4]: firstlast["title"] = firstlast["string"].str.title()

In [5]: firstlast
Out[5]:
       string       upper       lower       title
0  John Smith  JOHN SMITH  john smith  John Smith
1   Jane Cook   JANE COOK   jane cook   Jane Cook
```

Merging

The following tables will be used in the merge examples:

```
In [1]: df1 = pd.DataFrame({"key": ["A", "B", "C", "D"], "value": np.random.randn(4)})

In [2]: df1
Out[2]:
  key     value
0   A  0.469112
1   B -0.282863
2   C -1.509059
3   D -1.135632

In [3]: df2 = pd.DataFrame({"key": ["B", "D", "D", "E"], "value": np.random.randn(4)})

In [4]: df2
Out[4]:
  key     value
0   B  1.212112
1   D -0.173215
2   D  0.119209
3   E -1.044236
```

In SAS, data must be explicitly sorted before merging. Different types of joins are accomplished using the `in=` dummy variables to track whether a match was found in one or both input frames.

```
proc sort data=df1;
    by key;
run;

proc sort data=df2;
    by key;
run;

data left_join inner_join right_join outer_join;
    merge df1(in=a) df2(in=b);
```

(continues on next page)

(continued from previous page)

```
    if a and b then output inner_join;
    if a then output left_join;
    if b then output right_join;
    if a or b then output outer_join;
run;
```

pandas DataFrames have a *merge()* method, which provides similar functionality. The data does not have to be sorted ahead of time, and different join types are accomplished via the how keyword.

```
In [1]: inner_join = df1.merge(df2, on=["key"], how="inner")

In [2]: inner_join
Out[2]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209

In [3]: left_join = df1.merge(df2, on=["key"], how="left")

In [4]: left_join
Out[4]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209

In [5]: right_join = df1.merge(df2, on=["key"], how="right")

In [6]: right_join
Out[6]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209
3   E       NaN -1.044236

In [7]: outer_join = df1.merge(df2, on=["key"], how="outer")

In [8]: outer_join
Out[8]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236
```

Missing data

Both pandas and SAS have a representation for missing data.

pandas represents missing data with the special float value `NaN` (not a number). Many of the semantics are the same; for example missing data propagates through numeric operations, and is ignored by default for aggregations.

```
In [1]: outer_join
Out[1]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236

In [2]: outer_join["value_x"] + outer_join["value_y"]
Out[2]:
0         NaN
1    0.929249
2         NaN
3   -1.308847
4   -1.016424
5         NaN
dtype: float64

In [3]: outer_join["value_x"].sum()
Out[3]: -3.5940742896293765
```

One difference is that missing data cannot be compared to its sentinel value. For example, in SAS you could do this to filter missing values.

```
data outer_join_nulls;
    set outer_join;
    if value_x = .;
run;

data outer_join_no_nulls;
    set outer_join;
    if value_x ^= .;
run;
```

In pandas, *`Series.isna()`* and *`Series.notna()`* can be used to filter the rows.

```
In [1]: outer_join[outer_join["value_x"].isna()]
Out[1]:
  key  value_x   value_y
5   E      NaN -1.044236

In [2]: outer_join[outer_join["value_x"].notna()]
Out[2]:
  key   value_x   value_y
0   A  0.469112       NaN
```

(continues on next page)

(continued from previous page)

```
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
```

pandas provides *a variety of methods to work with missing data*. Here are some examples:

Drop rows with missing values

```
In [3]: outer_join.dropna()
Out[3]:
  key   value_x   value_y
1   B -0.282863  1.212112
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
```

Forward fill from previous rows

```
In [4]: outer_join.fillna(method="ffill")
Out[4]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059  1.212112
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E -1.135632 -1.044236
```

Replace missing values with a specified value

Using the mean:

```
In [1]: outer_join["value_x"].fillna(outer_join["value_x"].mean())
Out[1]:
0    0.469112
1   -0.282863
2   -1.509059
3   -1.135632
4   -1.135632
5   -0.718815
Name: value_x, dtype: float64
```

GroupBy

Aggregation

SAS's `PROC SUMMARY` can be used to group by one or more key variables and compute aggregations on numeric columns.

```
proc summary data=tips nway;
    class sex smoker;
    var total_bill tip;
    output out=tips_summed sum=;
run;
```

pandas provides a flexible `groupby` mechanism that allows similar aggregations. See the *groupby documentation* for more details and examples.

```
In [1]: tips_summed = tips.groupby(["sex", "smoker"])[["total_bill", "tip"]].sum()

In [2]: tips_summed
Out[2]:
               total_bill     tip
sex    smoker
Female No          869.68  149.77
       Yes         527.27   96.74
Male   No         1725.75  302.00
       Yes        1217.07  183.07
```

Transformation

In SAS, if the group aggregations need to be used with the original frame, it must be merged back together. For example, to subtract the mean for each observation by smoker group.

```
proc summary data=tips missing nway;
    class smoker;
    var total_bill;
    output out=smoker_means mean(total_bill)=group_bill;
run;

proc sort data=tips;
    by smoker;
run;

data tips;
    merge tips(in=a) smoker_means(in=b);
    by smoker;
    adj_total_bill = total_bill - group_bill;
    if a and b;
run;
```

pandas provides a *Transformation* mechanism that allows these type of operations to be succinctly expressed in one operation.

```
In [1]: gb = tips.groupby("smoker")["total_bill"]

In [2]: tips["adj_total_bill"] = tips["total_bill"] - gb.transform("mean")

In [3]: tips
Out[3]:
     total_bill    tip     sex smoker   day    time  size  adj_total_bill
67         1.07   1.00  Female    Yes   Sat  Dinner     1      -17.686344
92         3.75   1.00  Female    Yes   Fri  Dinner     2      -15.006344
111        5.25   1.00  Female     No   Sat  Dinner     1      -11.938278
145        6.35   1.50  Female     No  Thur   Lunch     2      -10.838278
135        6.51   1.25  Female     No  Thur   Lunch     2      -10.678278
..          ...    ...     ...    ...   ...     ...   ...             ...
182       43.35   3.50    Male    Yes   Sun  Dinner     3       24.593656
156       46.17   5.00    Male     No   Sun  Dinner     6       28.981722
59        46.27   6.73    Male     No   Sat  Dinner     4       29.081722
212       46.33   9.00    Male     No   Sat  Dinner     4       29.141722
170       48.81  10.00    Male    Yes   Sat  Dinner     3       30.053656

[244 rows x 8 columns]
```

By group processing

In addition to aggregation, pandas `groupby` can be used to replicate most other by group processing from SAS. For example, this `DATA` step reads the data by sex/smoker group and filters to the first entry for each.

```
proc sort data=tips;
   by sex smoker;
run;

data tips_first;
    set tips;
    by sex smoker;
    if FIRST.sex or FIRST.smoker then output;
run;
```

In pandas this would be written as:

```
In [4]: tips.groupby(["sex", "smoker"]).first()
Out[4]:
               total_bill   tip   day    time  size  adj_total_bill
sex    smoker
Female No            5.25  1.00   Sat  Dinner     1      -11.938278
       Yes           1.07  1.00   Sat  Dinner     1      -17.686344
Male   No            5.51  2.00  Thur   Lunch     2      -11.678278
       Yes           5.25  5.15   Sun  Dinner     2      -13.506344
```

Other considerations

Disk vs memory

pandas operates exclusively in memory, where a SAS data set exists on disk. This means that the size of data able to be loaded in pandas is limited by your machine's memory, but also that the operations on that data may be faster.

If out of core processing is needed, one possibility is the dask.dataframe library (currently in development) which provides a subset of pandas functionality for an on-disk `DataFrame`

Data interop

pandas provides a *read_sas()* method that can read SAS data saved in the XPORT or SAS7BDAT binary format.

```
libname xportout xport 'transport-file.xpt';
data xportout.tips;
    set tips(rename=(total_bill=tbill));
    * xport variable names limited to 6 characters;
run;
```

```
df = pd.read_sas("transport-file.xpt")
df = pd.read_sas("binary-file.sas7bdat")
```

You can also specify the file format directly. By default, pandas will try to infer the file format based on its extension.

```
df = pd.read_sas("transport-file.xpt", format="xport")
df = pd.read_sas("binary-file.sas7bdat", format="sas7bdat")
```

XPORT is a relatively limited format and the parsing of it is not as optimized as some of the other pandas readers. An alternative way to interop data between SAS and pandas is to serialize to csv.

```
# version 0.17, 10M rows

In [8]: %time df = pd.read_sas('big.xpt')
Wall time: 14.6 s

In [9]: %time df = pd.read_csv('big.csv')
Wall time: 4.86 s
```

Comparison with Stata

For potential users coming from Stata this page is meant to demonstrate how different Stata operations would be performed in pandas.

If you're new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

```
In [1]: import pandas as pd

In [2]: import numpy as np
```

Data structures

General terminology translation

pandas	Stata
DataFrame	data set
column	variable
row	observation
groupby	bysort
NaN	.

DataFrame

A DataFrame in pandas is analogous to a Stata data set – a two-dimensional data source with labeled columns that can be of different types. As will be shown in this document, almost any operation that can be applied to a data set in Stata can also be accomplished in pandas.

Series

A Series is the data structure that represents one column of a DataFrame. Stata doesn't have a separate data structure for a single column, but in general, working with a Series is analogous to referencing a column of a data set in Stata.

Index

Every DataFrame and Series has an Index – labels on the *rows* of the data. Stata does not have an exactly analogous concept. In Stata, a data set's rows are essentially unlabeled, other than an implicit integer index that can be accessed with _n.

In pandas, if no index is specified, an integer index is also used by default (first row = 0, second row = 1, and so on). While using a labeled Index or MultiIndex can enable sophisticated analyses and is ultimately an important part of pandas to understand, for this comparison we will essentially ignore the Index and just treat the DataFrame as a collection of columns. Please see the *indexing documentation* for much more on how to use an Index effectively.

Copies vs. in place operations

Most pandas operations return copies of the Series/DataFrame. To make the changes "stick", you'll need to either assign to a new variable:

```
sorted_df = df.sort_values("col1")
```

or overwrite the original one:

```
df = df.sort_values("col1")
```

Note: You will see an inplace=True keyword argument available for some methods:

```
df.sort_values("col1", inplace=True)
```

Its use is discouraged. *More information.*

Data input / output

Constructing a DataFrame from values

A Stata data set can be built from specified values by placing the data after an `input` statement and specifying the column names.

```
input x y
1 2
3 4
5 6
end
```

A pandas `DataFrame` can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

```
In [3]: df = pd.DataFrame({"x": [1, 3, 5], "y": [2, 4, 6]})

In [4]: df
Out[4]:
   x  y
0  1  2
1  3  4
2  5  6
```

Reading external data

Like Stata, pandas provides utilities for reading in data from many formats. The `tips` data set, found within the pandas tests (csv) will be used in many of the following examples.

Stata provides `import delimited` to read csv data into a data set in memory. If the `tips.csv` file is in the current working directory, we can import it as follows.

```
import delimited tips.csv
```

The pandas method is `read_csv()`, which works similarly. Additionally, it will automatically download the data set if presented with a url.

```
In [5]: url = (
   ...:     "https://raw.github.com/pandas-dev"
   ...:     "/pandas/main/pandas/tests/io/data/csv/tips.csv"
   ...: )
   ...:

In [6]: tips = pd.read_csv(url)

In [7]: tips
Out[7]:
     total_bill   tip     sex smoker   day    time  size
```

(continues on next page)

(continued from previous page)

```
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
4         24.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       29.03  5.92    Male     No   Sat  Dinner     3
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]
```

Like `import delimited`, *read_csv()* can take a number of parameters to specify how the data should be parsed. For example, if the data were instead tab delimited, did not have column names, and existed in the current working directory, the pandas command would be:

```
tips = pd.read_csv("tips.csv", sep="\t", header=None)

# alternatively, read_table is an alias to read_csv with tab delimiter
tips = pd.read_table("tips.csv", header=None)
```

pandas can also read Stata data sets in `.dta` format with the *read_stata()* function.

```
df = pd.read_stata("data.dta")
```

In addition to text/csv and Stata files, pandas supports a variety of other data formats such as Excel, SAS, HDF5, Parquet, and SQL databases. These are all read via a `pd.read_*` function. See the *IO documentation* for more details.

Limiting output

By default, pandas will truncate output of large `DataFrames` to show the first and last rows. This can be overridden by *changing the pandas options*, or using *DataFrame.head()* or *DataFrame.tail()*.

```
In [8]: tips.head(5)
Out[8]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

The equivalent in Stata would be:

```
list in 1/5
```

Exporting data

The inverse of `import delimited` in Stata is `export delimited`

```
export delimited tips2.csv
```

Similarly in pandas, the opposite of `read_csv` is *DataFrame.to_csv()*.

```
tips.to_csv("tips2.csv")
```

pandas can also export to Stata file format with the *DataFrame.to_stata()* method.

```
tips.to_stata("tips2.dta")
```

Data operations

Operations on columns

In Stata, arbitrary math expressions can be used with the `generate` and `replace` commands on new or existing columns. The `drop` command drops the column from the data set.

```
replace total_bill = total_bill - 2
generate new_bill = total_bill / 2
drop new_bill
```

pandas provides vectorized operations by specifying the individual `Series` in the `DataFrame`. New columns can be assigned in the same way. The *DataFrame.drop()* method drops a column from the `DataFrame`.

```
In [9]: tips["total_bill"] = tips["total_bill"] - 2

In [10]: tips["new_bill"] = tips["total_bill"] / 2

In [11]: tips
Out[11]:
     total_bill   tip     sex smoker   day    time  size  new_bill
0         14.99  1.01  Female     No   Sun  Dinner     2     7.495
1          8.34  1.66    Male     No   Sun  Dinner     3     4.170
2         19.01  3.50    Male     No   Sun  Dinner     3     9.505
3         21.68  3.31    Male     No   Sun  Dinner     2    10.840
4         22.59  3.61  Female     No   Sun  Dinner     4    11.295
..          ...   ...     ...    ...   ...     ...   ...       ...
239       27.03  5.92    Male     No   Sat  Dinner     3    13.515
240       25.18  2.00  Female    Yes   Sat  Dinner     2    12.590
241       20.67  2.00    Male    Yes   Sat  Dinner     2    10.335
242       15.82  1.75    Male     No   Sat  Dinner     2     7.910
243       16.78  3.00  Female     No  Thur  Dinner     2     8.390

[244 rows x 8 columns]

In [12]: tips = tips.drop("new_bill", axis=1)
```

Filtering

Filtering in Stata is done with an `if` clause on one or more columns.

```
list if total_bill > 10
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using *boolean indexing*.

```
In [13]: tips[tips["total_bill"] > 10]
Out[13]:
     total_bill   tip     sex smoker   day    time  size
0         14.99  1.01  Female     No   Sun  Dinner     2
2         19.01  3.50    Male     No   Sun  Dinner     3
3         21.68  3.31    Male     No   Sun  Dinner     2
4         22.59  3.61  Female     No   Sun  Dinner     4
5         23.29  4.71    Male     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       27.03  5.92    Male     No   Sat  Dinner     3
240       25.18  2.00  Female    Yes   Sat  Dinner     2
241       20.67  2.00    Male    Yes   Sat  Dinner     2
242       15.82  1.75    Male     No   Sat  Dinner     2
243       16.78  3.00  Female     No  Thur  Dinner     2

[204 rows x 7 columns]
```

The above statement is simply passing a `Series` of `True/False` objects to the DataFrame, returning all rows with `True`.

```
In [14]: is_dinner = tips["time"] == "Dinner"

In [15]: is_dinner
Out[15]:
0      True
1      True
2      True
3      True
4      True
       ...
239    True
240    True
241    True
242    True
243    True
Name: time, Length: 244, dtype: bool

In [16]: is_dinner.value_counts()
Out[16]:
True     176
False     68
Name: time, dtype: int64

In [17]: tips[is_dinner]
Out[17]:
```

(continues on next page)

(continued from previous page)

```
     total_bill   tip     sex smoker   day    time  size
0         14.99  1.01  Female     No   Sun  Dinner     2
1          8.34  1.66    Male     No   Sun  Dinner     3
2         19.01  3.50    Male     No   Sun  Dinner     3
3         21.68  3.31    Male     No   Sun  Dinner     2
4         22.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       27.03  5.92    Male     No   Sat  Dinner     3
240       25.18  2.00  Female    Yes   Sat  Dinner     2
241       20.67  2.00    Male    Yes   Sat  Dinner     2
242       15.82  1.75    Male     No   Sat  Dinner     2
243       16.78  3.00  Female     No  Thur  Dinner     2

[176 rows x 7 columns]
```

If/then logic

In Stata, an `if` clause can also be used to create new columns.

```
generate bucket = "low" if total_bill < 10
replace bucket = "high" if total_bill >= 10
```

The same operation in pandas can be accomplished using the `where` method from `numpy`.

```
In [18]: tips["bucket"] = np.where(tips["total_bill"] < 10, "low", "high")

In [19]: tips
Out[19]:
     total_bill   tip     sex smoker   day    time  size bucket
0         14.99  1.01  Female     No   Sun  Dinner     2   high
1          8.34  1.66    Male     No   Sun  Dinner     3    low
2         19.01  3.50    Male     No   Sun  Dinner     3   high
3         21.68  3.31    Male     No   Sun  Dinner     2   high
4         22.59  3.61  Female     No   Sun  Dinner     4   high
..          ...   ...     ...    ...   ...     ...   ...    ...
239       27.03  5.92    Male     No   Sat  Dinner     3   high
240       25.18  2.00  Female    Yes   Sat  Dinner     2   high
241       20.67  2.00    Male    Yes   Sat  Dinner     2   high
242       15.82  1.75    Male     No   Sat  Dinner     2   high
243       16.78  3.00  Female     No  Thur  Dinner     2   high

[244 rows x 8 columns]
```

Date functionality

Stata provides a variety of functions to do operations on date/datetime columns.

```
generate date1 = mdy(1, 15, 2013)
generate date2 = date("Feb152015", "MDY")

generate date1_year = year(date1)
generate date2_month = month(date2)

* shift date to beginning of next month
generate date1_next = mdy(month(date1) + 1, 1, year(date1)) if month(date1) != 12
replace date1_next = mdy(1, 1, year(date1) + 1) if month(date1) == 12
generate months_between = mofd(date2) - mofd(date1)

list date1 date2 date1_year date2_month date1_next months_between
```

The equivalent pandas operations are shown below. In addition to these functions, pandas supports other Time Series features not available in Stata (such as time zone handling and custom offsets) – see the *timeseries documentation* for more details.

```
In [20]: tips["date1"] = pd.Timestamp("2013-01-15")

In [21]: tips["date2"] = pd.Timestamp("2015-02-15")

In [22]: tips["date1_year"] = tips["date1"].dt.year

In [23]: tips["date2_month"] = tips["date2"].dt.month

In [24]: tips["date1_next"] = tips["date1"] + pd.offsets.MonthBegin()

In [25]: tips["months_between"] = tips["date2"].dt.to_period("M") - tips[
   ....:     "date1"
   ....: ].dt.to_period("M")
   ....: 

In [26]: tips[
   ....:     ["date1", "date2", "date1_year", "date2_month", "date1_next", "months_
↪between"]
   ....: ]
   ....: 
Out[26]: 
         date1      date2  date1_year  date2_month date1_next    months_between
0   2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
1   2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
2   2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
3   2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
4   2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
..         ...        ...         ...          ...        ...               ...
239 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
240 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
241 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
242 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
```

(continues on next page)

(continued from previous page)

```
243 2013-01-15 2015-02-15        2013             2 2013-02-01  <25 * MonthEnds>

[244 rows x 6 columns]
```

Selection of columns

Stata provides keywords to select, drop, and rename columns.

```
keep sex total_bill tip

drop sex

rename total_bill total_bill_2
```

The same operations are expressed in pandas below.

Keep certain columns

```
In [27]: tips[["sex", "total_bill", "tip"]]
Out[27]:
        sex  total_bill   tip
0    Female       14.99  1.01
1      Male        8.34  1.66
2      Male       19.01  3.50
3      Male       21.68  3.31
4    Female       22.59  3.61
..      ...         ...   ...
239    Male       27.03  5.92
240  Female       25.18  2.00
241    Male       20.67  2.00
242    Male       15.82  1.75
243  Female       16.78  3.00

[244 rows x 3 columns]
```

Drop a column

```
In [28]: tips.drop("sex", axis=1)
Out[28]:
     total_bill   tip smoker   day    time  size
0         14.99  1.01     No   Sun  Dinner     2
1          8.34  1.66     No   Sun  Dinner     3
2         19.01  3.50     No   Sun  Dinner     3
3         21.68  3.31     No   Sun  Dinner     2
4         22.59  3.61     No   Sun  Dinner     4
..          ...   ...    ...   ...     ...   ...
239       27.03  5.92     No   Sat  Dinner     3
240       25.18  2.00    Yes   Sat  Dinner     2
```

(continues on next page)

(continued from previous page)

```
241       20.67  2.00    Yes   Sat  Dinner     2
242       15.82  1.75     No   Sat  Dinner     2
243       16.78  3.00     No  Thur  Dinner     2

[244 rows x 6 columns]
```

Rename a column

```
In [29]: tips.rename(columns={"total_bill": "total_bill_2"})
Out[29]:
     total_bill_2   tip     sex smoker   day    time  size
0           14.99  1.01  Female     No   Sun  Dinner     2
1            8.34  1.66    Male     No   Sun  Dinner     3
2           19.01  3.50    Male     No   Sun  Dinner     3
3           21.68  3.31    Male     No   Sun  Dinner     2
4           22.59  3.61  Female     No   Sun  Dinner     4
..            ...   ...     ...    ...   ...     ...   ...
239         27.03  5.92    Male     No   Sat  Dinner     3
240         25.18  2.00  Female    Yes   Sat  Dinner     2
241         20.67  2.00    Male    Yes   Sat  Dinner     2
242         15.82  1.75    Male     No   Sat  Dinner     2
243         16.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]
```

Sorting by values

Sorting in Stata is accomplished via `sort`

```
sort sex total_bill
```

pandas has a `DataFrame.sort_values()` method, which takes a list of columns to sort by.

```
In [30]: tips = tips.sort_values(["sex", "total_bill"])

In [31]: tips
Out[31]:
     total_bill    tip     sex smoker   day    time  size
67         1.07   1.00  Female    Yes   Sat  Dinner     1
92         3.75   1.00  Female    Yes   Fri  Dinner     2
111        5.25   1.00  Female     No   Sat  Dinner     1
145        6.35   1.50  Female     No  Thur   Lunch     2
135        6.51   1.25  Female     No  Thur   Lunch     2
..          ...    ...     ...    ...   ...     ...   ...
182       43.35   3.50    Male    Yes   Sun  Dinner     3
156       46.17   5.00    Male     No   Sun  Dinner     6
59        46.27   6.73    Male     No   Sat  Dinner     4
212       46.33   9.00    Male     No   Sat  Dinner     4
170       48.81  10.00    Male    Yes   Sat  Dinner     3
```

(continues on next page)

(continued from previous page)

```
[244 rows x 7 columns]
```

String processing

Finding length of string

Stata determines the length of a character string with the `strlen()` and `ustrlen()` functions for ASCII and Unicode strings, respectively.

```
generate strlen_time = strlen(time)
generate ustrlen_time = ustrlen(time)
```

You can find the length of a character string with *Series.str.len()*. In Python 3, all strings are Unicode strings. `len` includes trailing blanks. Use `len` and `rstrip` to exclude trailing blanks.

```
In [32]: tips["time"].str.len()
Out[32]:
67     6
92     6
111    6
145    5
135    5
      ..
182    6
156    6
59     6
212    6
170    6
Name: time, Length: 244, dtype: int64

In [33]: tips["time"].str.rstrip().str.len()
Out[33]:
67     6
92     6
111    6
145    5
135    5
      ..
182    6
156    6
59     6
212    6
170    6
Name: time, Length: 244, dtype: int64
```

Finding position of substring

Stata determines the position of a character in a string with the `strpos()` function. This takes the string defined by the first argument and searches for the first position of the substring you supply as the second argument.

```
generate str_position = strpos(sex, "ale")
```

You can find the position of a character in a column of strings with the *`Series.str.find()`* method. `find` searches for the first position of the substring. If the substring is found, the method returns its position. If not found, it returns `-1`. Keep in mind that Python indexes are zero-based.

```
In [34]: tips["sex"].str.find("ale")
Out[34]:
67     3
92     3
111    3
145    3
135    3
      ..
182    1
156    1
59     1
212    1
170    1
Name: sex, Length: 244, dtype: int64
```

Extracting substring by position

Stata extracts a substring from a string based on its position with the `substr()` function.

```
generate short_sex = substr(sex, 1, 1)
```

With pandas you can use `[]` notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [35]: tips["sex"].str[0:1]
Out[35]:
67     F
92     F
111    F
145    F
135    F
      ..
182    M
156    M
59     M
212    M
170    M
Name: sex, Length: 244, dtype: object
```

Extracting nth word

The Stata `word()` function returns the nth word from a string. The first argument is the string you want to parse and the second argument specifies which word you want to extract.

```
clear
input str20 string
"John Smith"
"Jane Cook"
end

generate first_name = word(name, 1)
generate last_name = word(name, -1)
```

The simplest way to extract words in pandas is to split the strings by spaces, then reference the word by index. Note there are more powerful approaches should you need them.

```
In [36]: firstlast = pd.DataFrame({"String": ["John Smith", "Jane Cook"]})

In [37]: firstlast["First_Name"] = firstlast["String"].str.split(" ", expand=True)[0]

In [38]: firstlast["Last_Name"] = firstlast["String"].str.rsplit(" ", expand=True)[1]

In [39]: firstlast
Out[39]:
       String First_Name Last_Name
0  John Smith       John     Smith
1   Jane Cook       Jane      Cook
```

Changing case

The Stata `strupper()`, `strlower()`, `strproper()`, `ustrupper()`, `ustrlower()`, and `ustrtitle()` functions change the case of ASCII and Unicode strings, respectively.

```
clear
input str20 string
"John Smith"
"Jane Cook"
end

generate upper = strupper(string)
generate lower = strlower(string)
generate title = strproper(string)
list
```

The equivalent pandas methods are *Series.str.upper()*, *Series.str.lower()*, and *Series.str.title()*.

```
In [40]: firstlast = pd.DataFrame({"string": ["John Smith", "Jane Cook"]})

In [41]: firstlast["upper"] = firstlast["string"].str.upper()

In [42]: firstlast["lower"] = firstlast["string"].str.lower()
```

(continues on next page)

(continued from previous page)

```
In [43]: firstlast["title"] = firstlast["string"].str.title()

In [44]: firstlast
Out[44]: 
       string       upper       lower       title
0  John Smith  JOHN SMITH  john smith  John Smith
1   Jane Cook   JANE COOK   jane cook   Jane Cook
```

Merging

The following tables will be used in the merge examples:

```
In [45]: df1 = pd.DataFrame({"key": ["A", "B", "C", "D"], "value": np.random.randn(4)})

In [46]: df1
Out[46]: 
  key     value
0   A  0.469112
1   B -0.282863
2   C -1.509059
3   D -1.135632

In [47]: df2 = pd.DataFrame({"key": ["B", "D", "D", "E"], "value": np.random.randn(4)})

In [48]: df2
Out[48]: 
  key     value
0   B  1.212112
1   D -0.173215
2   D  0.119209
3   E -1.044236
```

In Stata, to perform a merge, one data set must be in memory and the other must be referenced as a file name on disk. In contrast, Python must have both `DataFrames` already in memory.

By default, Stata performs an outer join, where all observations from both data sets are left in memory after the merge. One can keep only observations from the initial data set, the merged data set, or the intersection of the two by using the values created in the `_merge` variable.

```
* First create df2 and save to disk
clear
input str1 key
B
D
D
E
end
generate value = rnormal()
save df2.dta
```

(continues on next page)

(continued from previous page)

```
* Now create df1 in memory
clear
input str1 key
A
B
C
D
end
generate value = rnormal()

preserve

* Left join
merge 1:n key using df2.dta
keep if _merge == 1

* Right join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 2

* Inner join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 3

* Outer join
restore
merge 1:n key using df2.dta
```

pandas DataFrames have a *merge()* method, which provides similar functionality. The data does not have to be sorted ahead of time, and different join types are accomplished via the how keyword.

```
In [49]: inner_join = df1.merge(df2, on=["key"], how="inner")

In [50]: inner_join
Out[50]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209

In [51]: left_join = df1.merge(df2, on=["key"], how="left")

In [52]: left_join
Out[52]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
```

(continues on next page)

(continued from previous page)

```
In [53]: right_join = df1.merge(df2, on=["key"], how="right")

In [54]: right_join
Out[54]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209
3   E       NaN -1.044236

In [55]: outer_join = df1.merge(df2, on=["key"], how="outer")

In [56]: outer_join
Out[56]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236
```

Missing data

Both pandas and Stata have a representation for missing data.

pandas represents missing data with the special float value `NaN` (not a number). Many of the semantics are the same; for example missing data propagates through numeric operations, and is ignored by default for aggregations.

```
In [57]: outer_join
Out[57]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236

In [58]: outer_join["value_x"] + outer_join["value_y"]
Out[58]:
0         NaN
1    0.929249
2         NaN
3   -1.308847
4   -1.016424
5         NaN
dtype: float64

In [59]: outer_join["value_x"].sum()
```

(continues on next page)

(continued from previous page)

```
Out[59]: -3.5940742896293765
```

One difference is that missing data cannot be compared to its sentinel value. For example, in Stata you could do this to filter missing values.

```
* Keep missing values
list if value_x == .
* Keep non-missing values
list if value_x != .
```

In pandas, *Series.isna()* and *Series.notna()* can be used to filter the rows.

```
In [60]: outer_join[outer_join["value_x"].isna()]
Out[60]:
  key  value_x   value_y
5   E      NaN -1.044236

In [61]: outer_join[outer_join["value_x"].notna()]
Out[61]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
```

pandas provides *a variety of methods to work with missing data*. Here are some examples:

Drop rows with missing values

```
In [62]: outer_join.dropna()
Out[62]:
  key   value_x   value_y
1   B -0.282863  1.212112
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
```

Forward fill from previous rows

```
In [63]: outer_join.fillna(method="ffill")
Out[63]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059  1.212112
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E -1.135632 -1.044236
```

Replace missing values with a specified value

Using the mean:

```
In [64]: outer_join["value_x"].fillna(outer_join["value_x"].mean())
Out[64]:
0    0.469112
1   -0.282863
2   -1.509059
3   -1.135632
4   -1.135632
5   -0.718815
Name: value_x, dtype: float64
```

GroupBy

Aggregation

Stata's `collapse` can be used to group by one or more key variables and compute aggregations on numeric columns.

```
collapse (sum) total_bill tip, by(sex smoker)
```

pandas provides a flexible `groupby` mechanism that allows similar aggregations. See the *groupby documentation* for more details and examples.

```
In [65]: tips_summed = tips.groupby(["sex", "smoker"])[["total_bill", "tip"]].sum()

In [66]: tips_summed
Out[66]:
               total_bill     tip
sex    smoker
Female No          869.68  149.77
       Yes         527.27   96.74
Male   No         1725.75  302.00
       Yes        1217.07  183.07
```

Transformation

In Stata, if the group aggregations need to be used with the original data set, one would usually use `bysort` with `egen()`. For example, to subtract the mean for each observation by smoker group.

```
bysort sex smoker: egen group_bill = mean(total_bill)
generate adj_total_bill = total_bill - group_bill
```

pandas provides a *Transformation* mechanism that allows these type of operations to be succinctly expressed in one operation.

```
In [67]: gb = tips.groupby("smoker")["total_bill"]

In [68]: tips["adj_total_bill"] = tips["total_bill"] - gb.transform("mean")
```

(continues on next page)

(continued from previous page)

```
In [69]: tips
Out[69]:
     total_bill    tip     sex smoker   day    time  size  adj_total_bill
67         1.07   1.00  Female    Yes   Sat  Dinner     1      -17.686344
92         3.75   1.00  Female    Yes   Fri  Dinner     2      -15.006344
111        5.25   1.00  Female     No   Sat  Dinner     1      -11.938278
145        6.35   1.50  Female     No  Thur   Lunch     2      -10.838278
135        6.51   1.25  Female     No  Thur   Lunch     2      -10.678278
..          ...    ...     ...    ...   ...     ...   ...             ...
182       43.35   3.50    Male    Yes   Sun  Dinner     3       24.593656
156       46.17   5.00    Male     No   Sun  Dinner     6       28.981722
59        46.27   6.73    Male     No   Sat  Dinner     4       29.081722
212       46.33   9.00    Male     No   Sat  Dinner     4       29.141722
170       48.81  10.00    Male    Yes   Sat  Dinner     3       30.053656

[244 rows x 8 columns]
```

By group processing

In addition to aggregation, pandas `groupby` can be used to replicate most other `bysort` processing from Stata. For example, the following example lists the first observation in the current sort order by sex/smoker group.

```
bysort sex smoker: list if _n == 1
```

In pandas this would be written as:

```
In [70]: tips.groupby(["sex", "smoker"]).first()
Out[70]:
               total_bill   tip   day    time  size  adj_total_bill
sex    smoker
Female No            5.25  1.00   Sat  Dinner     1      -11.938278
       Yes           1.07  1.00   Sat  Dinner     1      -17.686344
Male   No            5.51  2.00  Thur   Lunch     2      -11.678278
       Yes           5.25  5.15   Sun  Dinner     2      -13.506344
```

Other considerations

Disk vs memory

pandas and Stata both operate exclusively in memory. This means that the size of data able to be loaded in pandas is limited by your machine's memory. If out of core processing is needed, one possibility is the dask.dataframe library, which provides a subset of pandas functionality for an on-disk `DataFrame`.

1.4.5 Community tutorials

This is a guide to many pandas tutorials by the community, geared mainly for new users.

pandas cookbook by Julia Evans

The goal of this 2015 cookbook (by Julia Evans) is to give you some concrete examples for getting started with pandas. These are examples with real-world data, and all the bugs and weirdness that entails. For the table of contents, see the pandas-cookbook GitHub repository.

pandas workshop by Stefanie Molin

An introductory workshop by Stefanie Molin designed to quickly get you up to speed with pandas using real-world datasets. It covers getting started with pandas, data wrangling, and data visualization (with some exposure to matplotlib and seaborn). The pandas-workshop GitHub repository features detailed environment setup instructions (including a Binder environment), slides and notebooks for following along, and exercises to practice the concepts. There is also a lab with new exercises on a dataset not covered in the workshop for additional practice.

Learn pandas by Hernan Rojas

A set of lesson for new pandas users: https://bitbucket.org/hrojas/learn-pandas

Practical data analysis with Python

This guide is an introduction to the data analysis process using the Python data ecosystem and an interesting open dataset. There are four sections covering selected topics as munging data, aggregating data, visualizing data and time series.

Exercises for new users

Practice your skills with real data sets and exercises. For more resources, please visit the main repository.

Modern pandas

Tutorial series written in 2016 by Tom Augspurger. The source may be found in the GitHub repository TomAugspurger/effective-pandas.

- Modern Pandas
- Method Chaining
- Indexes
- Performance
- Tidy Data
- Visualization
- Timeseries

Excel charts with pandas, vincent and xlsxwriter

- Using Pandas and XlsxWriter to create Excel charts

Video tutorials

- Pandas From The Ground Up (2015) (2:24) GitHub repo
- Introduction Into Pandas (2016) (1:28) GitHub repo
- Pandas: .head() to .tail() (2016) (1:26) GitHub repo
- Data analysis in Python with pandas (2016-2018) GitHub repo and Jupyter Notebook
- Best practices with pandas (2018) GitHub repo and Jupyter Notebook

Various tutorials

- Wes McKinney's (pandas BDFL) blog
- Statistical analysis made easy in Python with SciPy and pandas DataFrames, by Randal Olson
- Statistical Data Analysis in Python, tutorial videos, by Christopher Fonnesbeck from SciPy 2013
- Financial analysis in Python, by Thomas Wiecki
- Intro to pandas data structures, by Greg Reda
- Pandas and Python: Top 10, by Manish Amde
- Pandas DataFrames Tutorial, by Karlijn Willems
- A concise tutorial with real life examples

CHAPTER TWO

USER GUIDE

The User Guide covers all of pandas by topic area. Each of the subsections introduces a topic (such as "working with missing data"), and discusses how pandas approaches the problem, with many examples throughout.

Users brand-new to pandas should start with 10min.

For a high level summary of the pandas fundamentals, see *Intro to data structures* and *Essential basic functionality*.

Further information on any specific method can be obtained in the *API reference*. {{ header }}

2.1 10 minutes to pandas

This is a short introduction to pandas, geared mainly for new users. You can see more complex recipes in the *Cookbook*.

Customarily, we import as follows:

```
In [1]: import numpy as np

In [2]: import pandas as pd
```

2.1.1 Object creation

See the *Intro to data structures section*.

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

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

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

Creating a `DataFrame` by passing a NumPy array, with a datetime index and labeled columns:

```
In [5]: dates = pd.date_range("20130101", periods=6)

In [6]: dates
Out[6]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

In [7]: df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))

In [8]: df
Out[8]:
                   A         B         C         D
2013-01-01 -0.653442 -0.421932  0.275996  1.073489
2013-01-02 -1.894721 -0.004210 -0.330351 -0.138219
2013-01-03 -0.779262 -0.624902 -0.981295  2.426929
2013-01-04 -2.610644  0.384005  0.334856  0.620484
2013-01-05 -0.787270 -0.570057  1.269041 -0.114205
2013-01-06  1.232899 -1.845574  1.155729 -1.167158
```

Creating a DataFrame by passing a dictionary of objects that can be converted into a series-like structure:

```
In [9]: df2 = pd.DataFrame(
   ...:     {
   ...:         "A": 1.0,
   ...:         "B": pd.Timestamp("20130102"),
   ...:         "C": pd.Series(1, index=list(range(4)), dtype="float32"),
   ...:         "D": np.array([3] * 4, dtype="int32"),
   ...:         "E": pd.Categorical(["test", "train", "test", "train"]),
   ...:         "F": "foo",
   ...:     }
   ...: )
   ...:

In [10]: df2
Out[10]:
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3  train  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
```

The columns of the resulting DataFrame have different *dtypes*:

```
In [11]: df2.dtypes
Out[11]:
A           float64
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

If you're using IPython, tab completion for column names (as well as public attributes) is automatically enabled. Here's a subset of the attributes that will be completed:

```
In [12]: df2.<TAB>  # noqa: E225, E999
df2.A                  df2.bool
df2.abs                df2.boxplot
df2.add                df2.C
df2.add_prefix         df2.clip
df2.add_suffix         df2.columns
df2.align              df2.copy
df2.all                df2.count
df2.any                df2.combine
df2.append             df2.D
df2.apply              df2.describe
df2.applymap           df2.diff
df2.B                  df2.duplicated
```

As you can see, the columns `A`, `B`, `C`, and `D` are automatically tab completed. `E` and `F` are there as well; the rest of the attributes have been truncated for brevity.

2.1.2 Viewing data

See the *Basics section*.

Here is how to view the top and bottom rows of the frame:

```
In [13]: df.head()
Out[13]:
                   A         B         C         D
2013-01-01 -0.653442 -0.421932  0.275996  1.073489
2013-01-02 -1.894721 -0.004210 -0.330351 -0.138219
2013-01-03 -0.779262 -0.624902 -0.981295  2.426929
2013-01-04 -2.610644  0.384005  0.334856  0.620484
2013-01-05 -0.787270 -0.570057  1.269041 -0.114205

In [14]: df.tail(3)
Out[14]:
                   A         B         C         D
2013-01-04 -2.610644  0.384005  0.334856  0.620484
2013-01-05 -0.787270 -0.570057  1.269041 -0.114205
2013-01-06  1.232899 -1.845574  1.155729 -1.167158
```

Display the index, columns:

```
In [15]: df.index
Out[15]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

In [16]: df.columns
Out[16]: Index(['A', 'B', 'C', 'D'], dtype='object')
```

`DataFrame.to_numpy()` gives a NumPy representation of the underlying data. Note that this can be an expensive operation when your `DataFrame` has columns with different data types, which comes down to a fundamental difference between pandas and NumPy: **NumPy arrays have one dtype for the entire array, while pandas DataFrames have one dtype per column**. When you call `DataFrame.to_numpy()`, pandas will find the NumPy dtype that can hold *all* of the dtypes in the DataFrame. This may end up being `object`, which requires casting every value to a Python object.

For `df`, our `DataFrame` of all floating-point values, `DataFrame.to_numpy()` is fast and doesn't require copying data:

```
In [17]: df.to_numpy()
Out[17]:
array([[-0.65344243, -0.42193239,  0.27599634,  1.07348944],
       [-1.89472066, -0.00421027, -0.33035087, -0.13821908],
       [-0.77926181, -0.62490223, -0.98129545,  2.42692938],
       [-2.61064369,  0.38400498,  0.33485625,  0.62048436],
       [-0.78727008, -0.57005682,  1.26904075, -0.11420549],
       [ 1.23289917, -1.84557387,  1.155729  , -1.16715815]])
```

For `df2`, the `DataFrame` with multiple dtypes, `DataFrame.to_numpy()` is relatively expensive:

```
In [18]: df2.to_numpy()
Out[18]:
array([[1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'test', 'foo'],
       [1.0, Timestamp('2013-01-02 00:00:00'), 1.0, 3, 'train', 'foo']],
      dtype=object)
```

Note: `DataFrame.to_numpy()` does *not* include the index or column labels in the output.

`describe()` shows a quick statistic summary of your data:

```
In [19]: df.describe()
Out[19]:
              A         B         C         D
count  6.000000  6.000000  6.000000  6.000000
mean  -0.915407 -0.513778  0.287329  0.450220
std    1.307798  0.756744  0.861534  1.233336
min   -2.610644 -1.845574 -0.981295 -1.167158
25%   -1.617858 -0.611191 -0.178764 -0.132216
50%   -0.783266 -0.495995  0.305426  0.253139
75%   -0.684897 -0.108641  0.950511  0.960238
max    1.232899  0.384005  1.269041  2.426929
```

Transposing your data:

```
In [20]: df.T
Out[20]:
   2013-01-01  2013-01-02  2013-01-03  2013-01-04  2013-01-05  2013-01-06
A   -0.653442   -1.894721   -0.779262   -2.610644   -0.787270    1.232899
B   -0.421932   -0.004210   -0.624902    0.384005   -0.570057   -1.845574
C    0.275996   -0.330351   -0.981295    0.334856    1.269041    1.155729
D    1.073489   -0.138219    2.426929    0.620484   -0.114205   -1.167158
```

Sorting by an axis:

```
In [21]: df.sort_index(axis=1, ascending=False)
Out[21]:
                   D         C         B         A
2013-01-01  1.073489  0.275996 -0.421932 -0.653442
2013-01-02 -0.138219 -0.330351 -0.004210 -1.894721
2013-01-03  2.426929 -0.981295 -0.624902 -0.779262
2013-01-04  0.620484  0.334856  0.384005 -2.610644
2013-01-05 -0.114205  1.269041 -0.570057 -0.787270
2013-01-06 -1.167158  1.155729 -1.845574  1.232899
```

Sorting by values:

```
In [22]: df.sort_values(by="B")
Out[22]:
                   A         B         C         D
2013-01-06  1.232899 -1.845574  1.155729 -1.167158
2013-01-03 -0.779262 -0.624902 -0.981295  2.426929
2013-01-05 -0.787270 -0.570057  1.269041 -0.114205
2013-01-01 -0.653442 -0.421932  0.275996  1.073489
2013-01-02 -1.894721 -0.004210 -0.330351 -0.138219
2013-01-04 -2.610644  0.384005  0.334856  0.620484
```

2.1.3 Selection

Note: While standard Python / NumPy expressions for selecting and setting are intuitive and come in handy for interactive work, for production code, we recommend the optimized pandas data access methods, `.at`, `.iat`, `.loc` and `.iloc`.

See the indexing documentation *Indexing and Selecting Data* and *MultiIndex / Advanced Indexing*.

Getting

Selecting a single column, which yields a `Series`, equivalent to `df.A`:

```
In [23]: df["A"]
Out[23]:
2013-01-01   -0.653442
2013-01-02   -1.894721
2013-01-03   -0.779262
2013-01-04   -2.610644
2013-01-05   -0.787270
2013-01-06    1.232899
Freq: D, Name: A, dtype: float64
```

Selecting via `[]`, which slices the rows:

```
In [24]: df[0:3]
Out[24]:
                   A         B         C         D
2013-01-01 -0.653442 -0.421932  0.275996  1.073489
```

(continues on next page)

(continued from previous page)

```
2013-01-02 -1.894721 -0.004210 -0.330351 -0.138219
2013-01-03 -0.779262 -0.624902 -0.981295  2.426929

In [25]: df["20130102":"20130104"]
Out[25]:
                   A         B         C         D
2013-01-02 -1.894721 -0.004210 -0.330351 -0.138219
2013-01-03 -0.779262 -0.624902 -0.981295  2.426929
2013-01-04 -2.610644  0.384005  0.334856  0.620484
```

Selection by label

See more in *Selection by Label*.

For getting a cross section using a label:

```
In [26]: df.loc[dates[0]]
Out[26]:
A   -0.653442
B   -0.421932
C    0.275996
D    1.073489
Name: 2013-01-01 00:00:00, dtype: float64
```

Selecting on a multi-axis by label:

```
In [27]: df.loc[:, ["A", "B"]]
Out[27]:
                   A         B
2013-01-01 -0.653442 -0.421932
2013-01-02 -1.894721 -0.004210
2013-01-03 -0.779262 -0.624902
2013-01-04 -2.610644  0.384005
2013-01-05 -0.787270 -0.570057
2013-01-06  1.232899 -1.845574
```

Showing label slicing, both endpoints are *included*:

```
In [28]: df.loc["20130102":"20130104", ["A", "B"]]
Out[28]:
                   A         B
2013-01-02 -1.894721 -0.004210
2013-01-03 -0.779262 -0.624902
2013-01-04 -2.610644  0.384005
```

Reduction in the dimensions of the returned object:

```
In [29]: df.loc["20130102", ["A", "B"]]
Out[29]:
A   -1.894721
B   -0.004210
Name: 2013-01-02 00:00:00, dtype: float64
```

For getting a scalar value:

```
In [30]: df.loc[dates[0], "A"]
Out[30]: -0.6534424311318969
```

For getting fast access to a scalar (equivalent to the prior method):

```
In [31]: df.at[dates[0], "A"]
Out[31]: -0.6534424311318969
```

Selection by position

See more in *Selection by Position*.

Select via the position of the passed integers:

```
In [32]: df.iloc[3]
Out[32]:
A   -2.610644
B    0.384005
C    0.334856
D    0.620484
Name: 2013-01-04 00:00:00, dtype: float64
```

By integer slices, acting similar to NumPy/Python:

```
In [33]: df.iloc[3:5, 0:2]
Out[33]:
                   A         B
2013-01-04 -2.610644  0.384005
2013-01-05 -0.787270 -0.570057
```

By lists of integer position locations, similar to the NumPy/Python style:

```
In [34]: df.iloc[[1, 2, 4], [0, 2]]
Out[34]:
                   A         C
2013-01-02 -1.894721 -0.330351
2013-01-03 -0.779262 -0.981295
2013-01-05 -0.787270  1.269041
```

For slicing rows explicitly:

```
In [35]: df.iloc[1:3, :]
Out[35]:
                   A         B         C         D
2013-01-02 -1.894721 -0.004210 -0.330351 -0.138219
2013-01-03 -0.779262 -0.624902 -0.981295  2.426929
```

For slicing columns explicitly:

```
In [36]: df.iloc[:, 1:3]
Out[36]:
                   B         C
```

(continues on next page)

(continued from previous page)

```
2013-01-01 -0.421932  0.275996
2013-01-02 -0.004210 -0.330351
2013-01-03 -0.624902 -0.981295
2013-01-04  0.384005  0.334856
2013-01-05 -0.570057  1.269041
2013-01-06 -1.845574  1.155729
```

For getting a value explicitly:

```
In [37]: df.iloc[1, 1]
Out[37]: -0.004210267301082381
```

For getting fast access to a scalar (equivalent to the prior method):

```
In [38]: df.iat[1, 1]
Out[38]: -0.004210267301082381
```

Boolean indexing

Using a single column's values to select data:

```
In [39]: df[df["A"] > 0]
Out[39]: 
                   A         B         C         D
2013-01-06  1.232899 -1.845574  1.155729 -1.167158
```

Selecting values from a DataFrame where a boolean condition is met:

```
In [40]: df[df > 0]
Out[40]: 
                   A         B         C         D
2013-01-01       NaN       NaN  0.275996  1.073489
2013-01-02       NaN       NaN       NaN       NaN
2013-01-03       NaN       NaN       NaN  2.426929
2013-01-04       NaN  0.384005  0.334856  0.620484
2013-01-05       NaN       NaN  1.269041       NaN
2013-01-06  1.232899       NaN  1.155729       NaN
```

Using the `isin()` method for filtering:

```
In [41]: df2 = df.copy()

In [42]: df2["E"] = ["one", "one", "two", "three", "four", "three"]

In [43]: df2
Out[43]: 
                   A         B         C         D      E
2013-01-01 -0.653442 -0.421932  0.275996  1.073489    one
2013-01-02 -1.894721 -0.004210 -0.330351 -0.138219    one
2013-01-03 -0.779262 -0.624902 -0.981295  2.426929    two
2013-01-04 -2.610644  0.384005  0.334856  0.620484  three
2013-01-05 -0.787270 -0.570057  1.269041 -0.114205   four
```

(continues on next page)

(continued from previous page)

```
2013-01-06  1.232899 -1.845574  1.155729 -1.167158  three

In [44]: df2[df2["E"].isin(["two", "four"])]
Out[44]:
                   A         B         C         D     E
2013-01-03 -0.779262 -0.624902 -0.981295  2.426929   two
2013-01-05 -0.787270 -0.570057  1.269041 -0.114205  four
```

Setting

Setting a new column automatically aligns the data by the indexes:

```
In [45]: s1 = pd.Series([1, 2, 3, 4, 5, 6], index=pd.date_range("20130102", periods=6))

In [46]: s1
Out[46]:
2013-01-02    1
2013-01-03    2
2013-01-04    3
2013-01-05    4
2013-01-06    5
2013-01-07    6
Freq: D, dtype: int64

In [47]: df["F"] = s1
```

Setting values by label:

```
In [48]: df.at[dates[0], "A"] = 0
```

Setting values by position:

```
In [49]: df.iat[0, 1] = 0
```

Setting by assigning with a NumPy array:

```
In [50]: df.loc[:, "D"] = np.array([5] * len(df))
```

The result of the prior setting operations:

```
In [51]: df
Out[51]:
                   A         B         C  D    F
2013-01-01  0.000000  0.000000  0.275996  5  NaN
2013-01-02 -1.894721 -0.004210 -0.330351  5  1.0
2013-01-03 -0.779262 -0.624902 -0.981295  5  2.0
2013-01-04 -2.610644  0.384005  0.334856  5  3.0
2013-01-05 -0.787270 -0.570057  1.269041  5  4.0
2013-01-06  1.232899 -1.845574  1.155729  5  5.0
```

A `where` operation with setting:

```
In [52]: df2 = df.copy()

In [53]: df2[df2 > 0] = -df2

In [54]: df2
Out[54]:
                   A         B         C  D    F
2013-01-01  0.000000  0.000000 -0.275996 -5  NaN
2013-01-02 -1.894721 -0.004210 -0.330351 -5 -1.0
2013-01-03 -0.779262 -0.624902 -0.981295 -5 -2.0
2013-01-04 -2.610644 -0.384005 -0.334856 -5 -3.0
2013-01-05 -0.787270 -0.570057 -1.269041 -5 -4.0
2013-01-06 -1.232899 -1.845574 -1.155729 -5 -5.0
```

2.1.4 Missing data

pandas primarily uses the value `np.nan` to represent missing data. It is by default not included in computations. See the *Missing Data section*.

Reindexing allows you to change/add/delete the index on a specified axis. This returns a copy of the data:

```
In [55]: df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ["E"])

In [56]: df1.loc[dates[0] : dates[1], "E"] = 1

In [57]: df1
Out[57]:
                   A         B         C  D    F    E
2013-01-01  0.000000  0.000000  0.275996  5  NaN  1.0
2013-01-02 -1.894721 -0.004210 -0.330351  5  1.0  1.0
2013-01-03 -0.779262 -0.624902 -0.981295  5  2.0  NaN
2013-01-04 -2.610644  0.384005  0.334856  5  3.0  NaN
```

To drop any rows that have missing data:

```
In [58]: df1.dropna(how="any")
Out[58]:
                   A        B         C  D    F    E
2013-01-02 -1.894721 -0.00421 -0.330351  5  1.0  1.0
```

Filling missing data:

```
In [59]: df1.fillna(value=5)
Out[59]:
                   A         B         C  D    F    E
2013-01-01  0.000000  0.000000  0.275996  5  5.0  1.0
2013-01-02 -1.894721 -0.004210 -0.330351  5  1.0  1.0
2013-01-03 -0.779262 -0.624902 -0.981295  5  2.0  5.0
2013-01-04 -2.610644  0.384005  0.334856  5  3.0  5.0
```

To get the boolean mask where values are `nan`:

```
In [60]: pd.isna(df1)
Out[60]:
                A      B      C      D      F      E
2013-01-01  False  False  False  False   True  False
2013-01-02  False  False  False  False  False  False
2013-01-03  False  False  False  False  False   True
2013-01-04  False  False  False  False  False   True
```

2.1.5 Operations

See the *Basic section on Binary Ops*.

Stats

Operations in general *exclude* missing data.

Performing a descriptive statistic:

```
In [61]: df.mean()
Out[61]:
A   -0.806500
B   -0.443456
C    0.287329
D    5.000000
F    3.000000
dtype: float64
```

Same operation on the other axis:

```
In [62]: df.mean(1)
Out[62]:
2013-01-01    1.318999
2013-01-02    0.754144
2013-01-03    0.922908
2013-01-04    1.221644
2013-01-05    1.782343
2013-01-06    2.108611
Freq: D, dtype: float64
```

Operating with objects that have different dimensionality and need alignment. In addition, pandas automatically broadcasts along the specified dimension:

```
In [63]: s = pd.Series([1, 3, 5, np.nan, 6, 8], index=dates).shift(2)

In [64]: s
Out[64]:
2013-01-01    NaN
2013-01-02    NaN
2013-01-03    1.0
2013-01-04    3.0
2013-01-05    5.0
2013-01-06    NaN
```

(continues on next page)

(continued from previous page)

```
Freq: D, dtype: float64

In [65]: df.sub(s, axis="index")
Out[65]:
                   A         B         C    D    F
2013-01-01       NaN       NaN       NaN  NaN  NaN
2013-01-02       NaN       NaN       NaN  NaN  NaN
2013-01-03 -1.779262 -1.624902 -1.981295  4.0  1.0
2013-01-04 -5.610644 -2.615995 -2.665144  2.0  0.0
2013-01-05 -5.787270 -5.570057 -3.730959  0.0 -1.0
2013-01-06       NaN       NaN       NaN  NaN  NaN
```

Apply

Applying functions to the data:

```
In [66]: df.apply(np.cumsum)
Out[66]:
                   A         B         C   D     F
2013-01-01  0.000000  0.000000  0.275996   5   NaN
2013-01-02 -1.894721 -0.004210 -0.054355  10   1.0
2013-01-03 -2.673982 -0.629112 -1.035650  15   3.0
2013-01-04 -5.284626 -0.245108 -0.700794  20   6.0
2013-01-05 -6.071896 -0.815164  0.568247  25  10.0
2013-01-06 -4.838997 -2.660738  1.723976  30  15.0

In [67]: df.apply(lambda x: x.max() - x.min())
Out[67]:
A    3.843543
B    2.229579
C    2.250336
D    0.000000
F    4.000000
dtype: float64
```

Histogramming

See more at *Histogramming and Discretization*.

```
In [68]: s = pd.Series(np.random.randint(0, 7, size=10))

In [69]: s
Out[69]:
0    1
1    2
2    0
3    6
4    1
5    2
6    1
```

(continues on next page)

(continued from previous page)

```
7    0
8    3
9    0
dtype: int64

In [70]: s.value_counts()
Out[70]:
1    3
0    3
2    2
6    1
3    1
dtype: int64
```

String Methods

Series is equipped with a set of string processing methods in the `str` attribute that make it easy to operate on each element of the array, as in the code snippet below. Note that pattern-matching in `str` generally uses regular expressions by default (and in some cases always uses them). See more at *Vectorized String Methods*.

```
In [71]: s = pd.Series(["A", "B", "C", "Aaba", "Baca", np.nan, "CABA", "dog", "cat"])

In [72]: s.str.lower()
Out[72]:
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

2.1.6 Merge

Concat

pandas provides various facilities for easily combining together Series and DataFrame objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

See the *Merging section*.

Concatenating pandas objects together with `concat()`:

```
In [73]: df = pd.DataFrame(np.random.randn(10, 4))

In [74]: df
Out[74]:
          0         1         2         3
```

(continues on next page)

(continued from previous page)

```
0 -0.126073 -2.456073  0.072982 -0.786972
1 -0.335904  0.515452  1.954611 -0.259685
2  0.016599  0.260267 -0.403736 -0.551094
3 -0.617292 -0.963827  0.188508  0.808858
4 -1.037205  0.372509 -0.155513  1.343315
5 -0.405051  0.132867  0.234965  1.279952
6  1.088834 -1.318023  0.680449  0.876654
7  0.728147 -0.682149 -1.449546  0.697394
8  1.129898  1.958106 -0.759490 -0.381220
9  0.841668 -1.142013  0.063833 -0.230865

# break it into pieces
In [75]: pieces = [df[:3], df[3:7], df[7:]]

In [76]: pd.concat(pieces)
Out[76]: 
          0         1         2         3
0 -0.126073 -2.456073  0.072982 -0.786972
1 -0.335904  0.515452  1.954611 -0.259685
2  0.016599  0.260267 -0.403736 -0.551094
3 -0.617292 -0.963827  0.188508  0.808858
4 -1.037205  0.372509 -0.155513  1.343315
5 -0.405051  0.132867  0.234965  1.279952
6  1.088834 -1.318023  0.680449  0.876654
7  0.728147 -0.682149 -1.449546  0.697394
8  1.129898  1.958106 -0.759490 -0.381220
9  0.841668 -1.142013  0.063833 -0.230865
```

Note: Adding a column to a DataFrame is relatively fast. However, adding a row requires a copy, and may be expensive. We recommend passing a pre-built list of records to the DataFrame constructor instead of building a DataFrame by iteratively appending records to it.

Join

SQL style merges. See the *Database style joining* section.

```
In [77]: left = pd.DataFrame({"key": ["foo", "foo"], "lval": [1, 2]})

In [78]: right = pd.DataFrame({"key": ["foo", "foo"], "rval": [4, 5]})

In [79]: left
Out[79]: 
   key  lval
0  foo     1
1  foo     2

In [80]: right
Out[80]: 
   key  rval
```

(continues on next page)

(continued from previous page)

```
0  foo     4
1  foo     5

In [81]: pd.merge(left, right, on="key")
Out[81]:
   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5
```

Another example that can be given is:

```
In [82]: left = pd.DataFrame({"key": ["foo", "bar"], "lval": [1, 2]})

In [83]: right = pd.DataFrame({"key": ["foo", "bar"], "rval": [4, 5]})

In [84]: left
Out[84]:
   key  lval
0  foo     1
1  bar     2

In [85]: right
Out[85]:
   key  rval
0  foo     4
1  bar     5

In [86]: pd.merge(left, right, on="key")
Out[86]:
   key  lval  rval
0  foo     1     4
1  bar     2     5
```

2.1.7 Grouping

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

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

See the *Grouping section*.

```
In [87]: df = pd.DataFrame(
   ....:     {
   ....:         "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
   ....:         "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
   ....:         "C": np.random.randn(8),
   ....:         "D": np.random.randn(8),
```

(continues on next page)

(continued from previous page)

```
   ....:     }
   ....: )
   ....:

In [88]: df
Out[88]:
     A      B         C         D
0  foo    one -1.421657 -0.187364
1  bar    one  0.510471 -0.162119
2  foo    two -0.575554 -0.145406
3  bar  three  0.127329  1.471945
4  foo    two  1.279711 -0.164349
5  bar    two -2.371887 -0.677948
6  foo    one  0.366695  1.101410
7  foo  three -1.220049  0.515292
```

Grouping and then applying the *sum()* function to the resulting groups:

```
In [89]: df.groupby("A").sum()
Out[89]:
            C         D
A
bar -1.734087  0.631878
foo -1.570854  1.119583
```

Grouping by multiple columns forms a hierarchical index, and again we can apply the *sum()* function:

```
In [90]: df.groupby(["A", "B"]).sum()
Out[90]:
                  C         D
A   B
bar one    0.510471 -0.162119
    three  0.127329  1.471945
    two   -2.371887 -0.677948
foo one   -1.054962  0.914046
    three -1.220049  0.515292
    two    0.704157 -0.309755
```

2.1.8 Reshaping

See the sections on *Hierarchical Indexing* and *Reshaping*.

Stack

```
In [91]: tuples = list(
   ....:     zip(
   ....:         *[
   ....:             ["bar", "bar", "baz", "baz", "foo", "foo", "qux", "qux"],
   ....:             ["one", "two", "one", "two", "one", "two", "one", "two"],
   ....:         ]
   ....:     )
   ....: )
   ....:

In [92]: index = pd.MultiIndex.from_tuples(tuples, names=["first", "second"])

In [93]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=["A", "B"])

In [94]: df2 = df[:4]

In [95]: df2
Out[95]:
                     A         B
first second
bar   one     0.630488  1.264926
      two    -0.642291 -0.103750
baz   one    -0.528482 -0.248170
      two     1.780117 -1.073086
```

The `stack()` method "compresses" a level in the DataFrame's columns:

```
In [96]: stacked = df2.stack()

In [97]: stacked
Out[97]:
first  second
bar    one     A    0.630488
               B    1.264926
       two     A   -0.642291
               B   -0.103750
baz    one     A   -0.528482
               B   -0.248170
       two     A    1.780117
               B   -1.073086
dtype: float64
```

With a "stacked" DataFrame or Series (having a `MultiIndex` as the `index`), the inverse operation of `stack()` is `unstack()`, which by default unstacks the **last level**:

```
In [98]: stacked.unstack()
Out[98]:
                     A         B
first second
bar   one     0.630488  1.264926
      two    -0.642291 -0.103750
```

(continues on next page)

(continued from previous page)

```
baz   one    -0.528482 -0.248170
      two     1.780117 -1.073086

In [99]: stacked.unstack(1)
Out[99]:
second         one       two
first
bar   A  0.630488 -0.642291
      B  1.264926 -0.103750
baz   A -0.528482  1.780117
      B -0.248170 -1.073086

In [100]: stacked.unstack(0)
Out[100]:
first          bar       baz
second
one    A  0.630488 -0.528482
       B  1.264926 -0.248170
two    A -0.642291  1.780117
       B -0.103750 -1.073086
```

Pivot tables

See the section on *Pivot Tables*.

```
In [101]: df = pd.DataFrame(
   .....:     {
   .....:         "A": ["one", "one", "two", "three"] * 3,
   .....:         "B": ["A", "B", "C"] * 4,
   .....:         "C": ["foo", "foo", "foo", "bar", "bar", "bar"] * 2,
   .....:         "D": np.random.randn(12),
   .....:         "E": np.random.randn(12),
   .....:     }
   .....: )
   .....:

In [102]: df
Out[102]:
        A  B    C         D         E
0     one  A  foo -1.320665  0.796906
1     one  B  foo  0.505476 -0.158080
2     two  C  foo  0.802241  0.022179
3   three  A  bar -0.061934  1.205095
4     one  B  bar  0.397417  0.757300
5     one  C  bar -1.386118 -0.378402
6     two  A  foo -0.363929  0.642860
7   three  B  foo  0.010883  1.178705
8     one  C  foo  1.310155 -0.685686
9     one  A  bar -0.277216 -0.097067
10    two  B  bar  0.676357 -0.718356
11  three  C  bar  1.623570 -1.967067
```

We can produce pivot tables from this data very easily:

```
In [103]: pd.pivot_table(df, values="D", index=["A", "B"], columns=["C"])
Out[103]:
C             bar       foo
A     B
one   A -0.277216 -1.320665
      B  0.397417  0.505476
      C -1.386118  1.310155
three A -0.061934       NaN
      B       NaN  0.010883
      C  1.623570       NaN
two   A       NaN -0.363929
      B  0.676357       NaN
      C       NaN  0.802241
```

2.1.9 Time series

pandas has simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (e.g., converting secondly data into 5-minutely data). This is extremely common in, but not limited to, financial applications. See the *Time Series section*.

```
In [104]: rng = pd.date_range("1/1/2012", periods=100, freq="S")

In [105]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

In [106]: ts.resample("5Min").sum()
Out[106]:
2012-01-01    24421
Freq: 5T, dtype: int64
```

Time zone representation:

```
In [107]: rng = pd.date_range("3/6/2012 00:00", periods=5, freq="D")

In [108]: ts = pd.Series(np.random.randn(len(rng)), rng)

In [109]: ts
Out[109]:
2012-03-06    0.861725
2012-03-07   -0.639643
2012-03-08    1.220722
2012-03-09    0.846082
2012-03-10    0.676839
Freq: D, dtype: float64

In [110]: ts_utc = ts.tz_localize("UTC")

In [111]: ts_utc
Out[111]:
2012-03-06 00:00:00+00:00    0.861725
2012-03-07 00:00:00+00:00   -0.639643
```

(continues on next page)

(continued from previous page)

```
2012-03-08 00:00:00+00:00    1.220722
2012-03-09 00:00:00+00:00    0.846082
2012-03-10 00:00:00+00:00    0.676839
Freq: D, dtype: float64
```

Converting to another time zone:

```
In [112]: ts_utc.tz_convert("US/Eastern")
Out[112]:
2012-03-05 19:00:00-05:00    0.861725
2012-03-06 19:00:00-05:00   -0.639643
2012-03-07 19:00:00-05:00    1.220722
2012-03-08 19:00:00-05:00    0.846082
2012-03-09 19:00:00-05:00    0.676839
Freq: D, dtype: float64
```

Converting between time span representations:

```
In [113]: rng = pd.date_range("1/1/2012", periods=5, freq="M")

In [114]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [115]: ts
Out[115]:
2012-01-31   -0.755355
2012-02-29    0.127516
2012-03-31    0.359423
2012-04-30   -0.157994
2012-05-31   -0.280931
Freq: M, dtype: float64

In [116]: ps = ts.to_period()

In [117]: ps
Out[117]:
2012-01   -0.755355
2012-02    0.127516
2012-03    0.359423
2012-04   -0.157994
2012-05   -0.280931
Freq: M, dtype: float64

In [118]: ps.to_timestamp()
Out[118]:
2012-01-01   -0.755355
2012-02-01    0.127516
2012-03-01    0.359423
2012-04-01   -0.157994
2012-05-01   -0.280931
Freq: MS, dtype: float64
```

Converting between period and timestamp enables some convenient arithmetic functions to be used. In the following example, we convert a quarterly frequency with year ending in November to 9am of the end of the month following the quarter end:

```
In [119]: prng = pd.period_range("1990Q1", "2000Q4", freq="Q-NOV")

In [120]: ts = pd.Series(np.random.randn(len(prng)), prng)

In [121]: ts.index = (prng.asfreq("M", "e") + 1).asfreq("H", "s") + 9

In [122]: ts.head()
Out[122]: 
1990-03-01 09:00    1.180210
1990-06-01 09:00   -0.815243
1990-09-01 09:00    1.024747
1990-12-01 09:00   -0.027438
1991-03-01 09:00   -0.180342
Freq: H, dtype: float64
```

2.1.10 Categoricals

pandas can include categorical data in a `DataFrame`. For full docs, see the *categorical introduction* and the *API documentation*.

```
In [123]: df = pd.DataFrame(
   .....:     {"id": [1, 2, 3, 4, 5, 6], "raw_grade": ["a", "b", "b", "a", "a", "e"]}
   .....: )
   .....: 
```

Converting the raw grades to a categorical data type:

```
In [124]: df["grade"] = df["raw_grade"].astype("category")

In [125]: df["grade"]
Out[125]: 
0    a
1    b
2    b
3    a
4    a
5    e
Name: grade, dtype: category
Categories (3, object): ['a', 'b', 'e']
```

Rename the categories to more meaningful names (assigning to `Series.cat.categories()` is in place!):

```
In [126]: df["grade"].cat.categories = ["very good", "good", "very bad"]
```

Reorder the categories and simultaneously add the missing categories (methods under `Series.cat()` return a new `Series` by default):

```
In [127]: df["grade"] = df["grade"].cat.set_categories(
   .....:     ["very bad", "bad", "medium", "good", "very good"]
   .....: )
   .....: 
```

(continues on next page)

(continued from previous page)

```
In [128]: df["grade"]
Out[128]:
0    very good
1         good
2         good
3    very good
4    very good
5     very bad
Name: grade, dtype: category
Categories (5, object): ['very bad', 'bad', 'medium', 'good', 'very good']
```

Sorting is per order in the categories, not lexical order:

```
In [129]: df.sort_values(by="grade")
Out[129]:
   id raw_grade      grade
5   6         e   very bad
1   2         b       good
2   3         b       good
0   1         a  very good
3   4         a  very good
4   5         a  very good
```

Grouping by a categorical column also shows empty categories:

```
In [130]: df.groupby("grade").size()
Out[130]:
grade
very bad     1
bad          0
medium       0
good         2
very good    3
dtype: int64
```

2.1.11 Plotting

See the *Plotting* docs.

We use the standard convention for referencing the matplotlib API:

```
In [131]: import matplotlib.pyplot as plt

In [132]: plt.close("all")
```

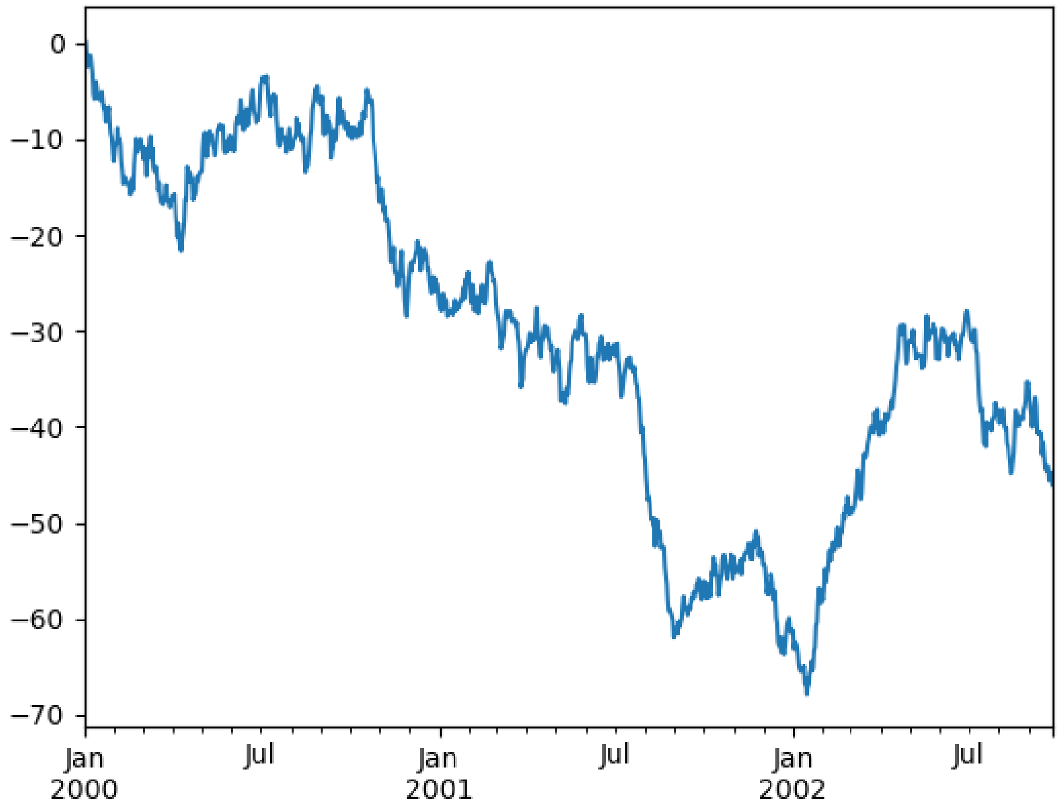
The `close()` method is used to close a figure window:

```
In [133]: ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000",␣
↪periods=1000))

In [134]: ts = ts.cumsum()
```

(continues on next page)

(continued from previous page)

```
In [135]: ts.plot();
```

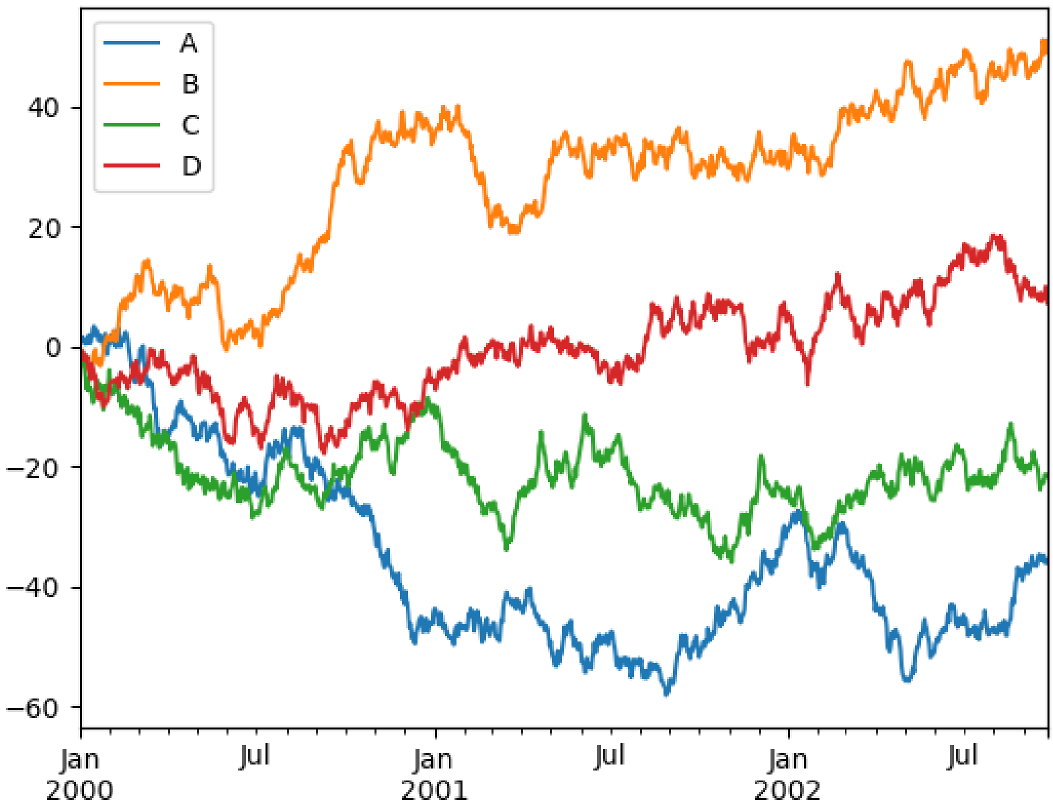


If running under Jupyter Notebook, the plot will appear on `plot()`. Otherwise use matplotlib.pyplot.show to show it or matplotlib.pyplot.savefig to write it to a file.

```
In [136]: plt.show();
```

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

```
In [137]: df = pd.DataFrame(
   .....:     np.random.randn(1000, 4), index=ts.index, columns=["A", "B", "C", "D"]
   .....: )
   .....:

In [138]: df = df.cumsum()

In [139]: plt.figure();

In [140]: df.plot();

In [141]: plt.legend(loc='best');
```



2.1.12 Getting data in/out

CSV

Writing to a csv file:

```
In [142]: df.to_csv("foo.csv")
```

Reading from a csv file:

```
In [143]: pd.read_csv("foo.csv")
Out[143]:
     Unnamed: 0          A          B          C          D
0    2000-01-01  -0.129121   1.916816   0.588168 -0.269016
1    2000-01-02   1.018676   1.112194   0.007985 -0.612719
2    2000-01-03   2.781368   2.255281  -1.156786 -0.186486
3    2000-01-04   3.477854   3.517590  -0.856852  0.516957
4    2000-01-05   4.816740   3.240747  -2.096812  0.641055
..          ...        ...        ...        ...        ...
995  2002-09-22  29.779576  12.090699  30.152225 -8.212937
996  2002-09-23  29.250386  11.489114  30.192422 -7.098868
997  2002-09-24  30.249793  11.326815  29.765853 -5.829072
```

(continues on next page)

(continued from previous page)

```
998   2002-09-25   32.801188   12.265137   28.941801 -5.852998
999   2002-09-26   32.442356   12.948902   29.717285 -6.719486

[1000 rows x 5 columns]
```

HDF5

Reading and writing to *HDFStores*.

Writing to a HDF5 Store:

```
In [144]: df.to_hdf("foo.h5", "df")
```

Reading from a HDF5 Store:

```
In [145]: pd.read_hdf("foo.h5", "df")
Out[145]: 
                    A          B          C         D
2000-01-01  -0.129121   1.916816   0.588168 -0.269016
2000-01-02   1.018676   1.112194   0.007985 -0.612719
2000-01-03   2.781368   2.255281  -1.156786 -0.186486
2000-01-04   3.477854   3.517590  -0.856852  0.516957
2000-01-05   4.816740   3.240747  -2.096812  0.641055
...               ...        ...        ...       ...
2002-09-22  29.779576  12.090699  30.152225 -8.212937
2002-09-23  29.250386  11.489114  30.192422 -7.098868
2002-09-24  30.249793  11.326815  29.765853 -5.829072
2002-09-25  32.801188  12.265137  28.941801 -5.852998
2002-09-26  32.442356  12.948902  29.717285 -6.719486

[1000 rows x 4 columns]
```

Excel

Reading and writing to *MS Excel*.

Writing to an excel file:

```
In [146]: df.to_excel("foo.xlsx", sheet_name="Sheet1")
```

Reading from an excel file:

```
In [147]: pd.read_excel("foo.xlsx", "Sheet1", index_col=None, na_values=["NA"])
Out[147]: 
    Unnamed: 0          A          B          C         D
0   2000-01-01  -0.129121   1.916816   0.588168 -0.269016
1   2000-01-02   1.018676   1.112194   0.007985 -0.612719
2   2000-01-03   2.781368   2.255281  -1.156786 -0.186486
3   2000-01-04   3.477854   3.517590  -0.856852  0.516957
4   2000-01-05   4.816740   3.240747  -2.096812  0.641055
..         ...        ...        ...        ...       ...
```

(continues on next page)

(continued from previous page)

```
995 2002-09-22  29.779576  12.090699  30.152225 -8.212937
996 2002-09-23  29.250386  11.489114  30.192422 -7.098868
997 2002-09-24  30.249793  11.326815  29.765853 -5.829072
998 2002-09-25  32.801188  12.265137  28.941801 -5.852998
999 2002-09-26  32.442356  12.948902  29.717285 -6.719486

[1000 rows x 5 columns]
```

2.1.13 Gotchas

If you are attempting to perform an operation you might see an exception like:

```
>>> if pd.Series([False, True, False]):
...     print("I was true")
Traceback
    ...
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

See *Comparisons* for an explanation and what to do.

See *Gotchas* as well.

2.2 Intro to data structures

We'll start with a quick, non-comprehensive overview of the fundamental data structures in pandas to get you started. The fundamental behavior about data types, indexing, and axis labeling / alignment apply across all of the objects. To get started, import NumPy and load pandas into your namespace:

```
In [1]: import numpy as np

In [2]: import pandas as pd
```

Here is a basic tenet to keep in mind: **data alignment is intrinsic**. The link between labels and data will not be broken unless done so explicitly by you.

We'll give a brief intro to the data structures, then consider all of the broad categories of functionality and methods in separate sections.

2.2.1 Series

`Series` is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the **index**. The basic method to create a Series is to call:

```
>>> s = pd.Series(data, index=index)
```

Here, `data` can be many different things:

- a Python dict
- an ndarray

- a scalar value (like 5)

The passed **index** is a list of axis labels. Thus, this separates into a few cases depending on what **data is**:

From ndarray

If `data` is an ndarray, **index** must be the same length as **data**. If no index is passed, one will be created having values `[0, ..., len(data) - 1]`.

```
In [3]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])

In [4]: s
Out[4]:
a    0.469112
b   -0.282863
c   -1.509059
d   -1.135632
e    1.212112
dtype: float64

In [5]: s.index
Out[5]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [6]: pd.Series(np.random.randn(5))
Out[6]:
0   -0.173215
1    0.119209
2   -1.044236
3   -0.861849
4   -2.104569
dtype: float64
```

Note: pandas supports non-unique index values. If an operation that does not support duplicate index values is attempted, an exception will be raised at that time. The reason for being lazy is nearly all performance-based (there are many instances in computations, like parts of GroupBy, where the index is not used).

From dict

Series can be instantiated from dicts:

```
In [7]: d = {"b": 1, "a": 0, "c": 2}

In [8]: pd.Series(d)
Out[8]:
b    1
a    0
c    2
dtype: int64
```

Note: When the data is a dict, and an index is not passed, the `Series` index will be ordered by the dict's insertion order, if you're using Python version >= 3.6 and pandas version >= 0.23.

If you're using Python < 3.6 or pandas < 0.23, and an index is not passed, the `Series` index will be the lexically ordered list of dict keys.

In the example above, if you were on a Python version lower than 3.6 or a pandas version lower than 0.23, the `Series` would be ordered by the lexical order of the dict keys (i.e. `['a', 'b', 'c']` rather than `['b', 'a', 'c']`).

If an index is passed, the values in data corresponding to the labels in the index will be pulled out.

```
In [9]: d = {"a": 0.0, "b": 1.0, "c": 2.0}

In [10]: pd.Series(d)
Out[10]:
a    0.0
b    1.0
c    2.0
dtype: float64

In [11]: pd.Series(d, index=["b", "c", "d", "a"])
Out[11]:
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Note: NaN (not a number) is the standard missing data marker used in pandas.

From scalar value

If `data` is a scalar value, an index must be provided. The value will be repeated to match the length of **index**.

```
In [12]: pd.Series(5.0, index=["a", "b", "c", "d", "e"])
Out[12]:
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Series is ndarray-like

`Series` acts very similarly to a `ndarray`, and is a valid argument to most NumPy functions. However, operations such as slicing will also slice the index.

```
In [13]: s[0]
Out[13]: 0.4691122999071863

In [14]: s[:3]
Out[14]:
a    0.469112
b   -0.282863
c   -1.509059
```

(continues on next page)

(continued from previous page)

```
dtype: float64

In [15]: s[s > s.median()]
Out[15]:
a    0.469112
e    1.212112
dtype: float64

In [16]: s[[4, 3, 1]]
Out[16]:
e    1.212112
d   -1.135632
b   -0.282863
dtype: float64

In [17]: np.exp(s)
Out[17]:
a    1.598575
b    0.753623
c    0.221118
d    0.321219
e    3.360575
dtype: float64
```

Note: We will address array-based indexing like `s[[4, 3, 1]]` in *section on indexing*.

Like a NumPy array, a pandas Series has a `dtype`.

```
In [18]: s.dtype
Out[18]: dtype('float64')
```

This is often a NumPy dtype. However, pandas and 3rd-party libraries extend NumPy's type system in a few places, in which case the dtype would be an `ExtensionDtype`. Some examples within pandas are *Categorical data* and *Nullable integer data type*. See *dtypes* for more.

If you need the actual array backing a `Series`, use `Series.array`.

```
In [19]: s.array
Out[19]:
<PandasArray>
[ 0.4691122999071863, -0.2828633443286633, -1.5090585031735124,
 -1.1356323710171934,  1.2121120250208506]
Length: 5, dtype: float64
```

Accessing the array can be useful when you need to do some operation without the index (to disable *automatic alignment*, for example).

`Series.array` will always be an `ExtensionArray`. Briefly, an ExtensionArray is a thin wrapper around one or more *concrete* arrays like a `numpy.ndarray`. pandas knows how to take an `ExtensionArray` and store it in a `Series` or a column of a `DataFrame`. See *dtypes* for more.

While Series is ndarray-like, if you need an *actual* ndarray, then use `Series.to_numpy()`.

```
In [20]: s.to_numpy()
Out[20]: array([ 0.4691, -0.2829, -1.5091, -1.1356,  1.2121])
```

Even if the Series is backed by a *ExtensionArray*, *Series.to_numpy()* will return a NumPy ndarray.

Series is dict-like

A Series is like a fixed-size dict in that you can get and set values by index label:

```
In [21]: s["a"]
Out[21]: 0.4691122999071863

In [22]: s["e"] = 12.0

In [23]: s
Out[23]:
a     0.469112
b    -0.282863
c    -1.509059
d    -1.135632
e    12.000000
dtype: float64

In [24]: "e" in s
Out[24]: True

In [25]: "f" in s
Out[25]: False
```

If a label is not contained, an exception is raised:

```
>>> s["f"]
KeyError: 'f'
```

Using the `get` method, a missing label will return None or specified default:

```
In [26]: s.get("f")

In [27]: s.get("f", np.nan)
Out[27]: nan
```

See also the *section on attribute access*.

Vectorized operations and label alignment with Series

When working with raw NumPy arrays, looping through value-by-value is usually not necessary. The same is true when working with Series in pandas. Series can also be passed into most NumPy methods expecting an ndarray.

```
In [28]: s + s
Out[28]:
a     0.938225
b    -0.565727
```

(continues on next page)

(continued from previous page)

```
c    -3.018117
d    -2.271265
e    24.000000
dtype: float64

In [29]: s * 2
Out[29]:
a     0.938225
b    -0.565727
c    -3.018117
d    -2.271265
e    24.000000
dtype: float64

In [30]: np.exp(s)
Out[30]:
a         1.598575
b         0.753623
c         0.221118
d         0.321219
e    162754.791419
dtype: float64
```

A key difference between Series and ndarray is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

```
In [31]: s[1:] + s[:-1]
Out[31]:
a         NaN
b   -0.565727
c   -3.018117
d   -2.271265
e         NaN
dtype: float64
```

The result of an operation between unaligned Series will have the **union** of the indexes involved. If a label is not found in one Series or the other, the result will be marked as missing `NaN`. Being able to write code without doing any explicit data alignment grants immense freedom and flexibility in interactive data analysis and research. The integrated data alignment features of the pandas data structures set pandas apart from the majority of related tools for working with labeled data.

Note: In general, we chose to make the default result of operations between differently indexed objects yield the **union** of the indexes in order to avoid loss of information. Having an index label, though the data is missing, is typically important information as part of a computation. You of course have the option of dropping labels with missing data via the **dropna** function.

Name attribute

Series can also have a `name` attribute:

```
In [32]: s = pd.Series(np.random.randn(5), name="something")

In [33]: s
Out[33]:
0   -0.494929
1    1.071804
2    0.721555
3   -0.706771
4   -1.039575
Name: something, dtype: float64

In [34]: s.name
Out[34]: 'something'
```

The Series `name` will be assigned automatically in many cases, in particular when taking 1D slices of DataFrame as you will see below.

You can rename a Series with the *pandas.Series.rename()* method.

```
In [35]: s2 = s.rename("different")

In [36]: s2.name
Out[36]: 'different'
```

Note that `s` and `s2` refer to different objects.

2.2.2 DataFrame

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object. Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A `Series`
- Another `DataFrame`

Along with the data, you can optionally pass **index** (row labels) and **columns** (column labels) arguments. If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.

If axis labels are not passed, they will be constructed from the input data based on common sense rules.

Note: When the data is a dict, and `columns` is not specified, the `DataFrame` columns will be ordered by the dict's insertion order, if you are using Python version >= 3.6 and pandas >= 0.23.

If you are using Python < 3.6 or pandas < 0.23, and `columns` is not specified, the `DataFrame` columns will be the lexically ordered list of dict keys.

From dict of Series or dicts

The resulting **index** will be the **union** of the indexes of the various Series. If there are any nested dicts, these will first be converted to Series. If no columns are passed, the columns will be the ordered list of dict keys.

```
In [37]: d = {
   ....:     "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"]),
   ....:     "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c", "d"]),
   ....: }
   ....:

In [38]: df = pd.DataFrame(d)

In [39]: df
Out[39]:
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

In [40]: pd.DataFrame(d, index=["d", "b", "a"])
Out[40]:
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0

In [41]: pd.DataFrame(d, index=["d", "b", "a"], columns=["two", "three"])
Out[41]:
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

The row and column labels can be accessed respectively by accessing the **index** and **columns** attributes:

Note: When a particular set of columns is passed along with a dict of data, the passed columns override the keys in the dict.

```
In [42]: df.index
Out[42]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [43]: df.columns
Out[43]: Index(['one', 'two'], dtype='object')
```

From dict of ndarrays / lists

The ndarrays must all be the same length. If an index is passed, it must clearly also be the same length as the arrays. If no index is passed, the result will be `range(n)`, where `n` is the array length.

```
In [44]: d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0, 1.0]}

In [45]: pd.DataFrame(d)
Out[45]:
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0

In [46]: pd.DataFrame(d, index=["a", "b", "c", "d"])
Out[46]:
   one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```

From structured or record array

This case is handled identically to a dict of arrays.

```
In [47]: data = np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "a10")])

In [48]: data[:] = [(1, 2.0, "Hello"), (2, 3.0, "World")]

In [49]: pd.DataFrame(data)
Out[49]:
   A    B         C
0  1  2.0  b'Hello'
1  2  3.0  b'World'

In [50]: pd.DataFrame(data, index=["first", "second"])
Out[50]:
        A    B         C
first   1  2.0  b'Hello'
second  2  3.0  b'World'

In [51]: pd.DataFrame(data, columns=["C", "A", "B"])
Out[51]:
          C  A    B
0  b'Hello'  1  2.0
1  b'World'  2  3.0
```

Note: DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.

From a list of dicts

```
In [52]: data2 = [{"a": 1, "b": 2}, {"a": 5, "b": 10, "c": 20}]

In [53]: pd.DataFrame(data2)
Out[53]:
   a   b     c
0  1   2   NaN
1  5  10  20.0

In [54]: pd.DataFrame(data2, index=["first", "second"])
Out[54]:
        a   b     c
first   1   2   NaN
second  5  10  20.0

In [55]: pd.DataFrame(data2, columns=["a", "b"])
Out[55]:
   a   b
0  1   2
1  5  10
```

From a dict of tuples

You can automatically create a MultiIndexed frame by passing a tuples dictionary.

```
In [56]: pd.DataFrame(
   ....:     {
   ....:         ("a", "b"): {("A", "B"): 1, ("A", "C"): 2},
   ....:         ("a", "a"): {("A", "C"): 3, ("A", "B"): 4},
   ....:         ("a", "c"): {("A", "B"): 5, ("A", "C"): 6},
   ....:         ("b", "a"): {("A", "C"): 7, ("A", "B"): 8},
   ....:         ("b", "b"): {("A", "D"): 9, ("A", "B"): 10},
   ....:     }
   ....: )
   ....:
Out[56]:
       a              b
       b    a    c    a     b
A B  1.0  4.0  5.0  8.0  10.0
  C  2.0  3.0  6.0  7.0   NaN
  D  NaN  NaN  NaN  NaN   9.0
```

From a Series

The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).

From a list of namedtuples

The field names of the first `namedtuple` in the list determine the columns of the `DataFrame`. The remaining namedtuples (or tuples) are simply unpacked and their values are fed into the rows of the `DataFrame`. If any of those tuples is shorter than the first `namedtuple` then the later columns in the corresponding row are marked as missing values. If any are longer than the first `namedtuple`, a `ValueError` is raised.

```
In [57]: from collections import namedtuple

In [58]: Point = namedtuple("Point", "x y")

In [59]: pd.DataFrame([Point(0, 0), Point(0, 3), (2, 3)])
Out[59]:
   x  y
0  0  0
1  0  3
2  2  3

In [60]: Point3D = namedtuple("Point3D", "x y z")

In [61]: pd.DataFrame([Point3D(0, 0, 0), Point3D(0, 3, 5), Point(2, 3)])
Out[61]:
   x  y    z
0  0  0  0.0
1  0  3  5.0
2  2  3  NaN
```

From a list of dataclasses

New in version 1.1.0.

Data Classes as introduced in PEP557, can be passed into the DataFrame constructor. Passing a list of dataclasses is equivalent to passing a list of dictionaries.

Please be aware, that all values in the list should be dataclasses, mixing types in the list would result in a TypeError.

```
In [62]: from dataclasses import make_dataclass

In [63]: Point = make_dataclass("Point", [("x", int), ("y", int)])

In [64]: pd.DataFrame([Point(0, 0), Point(0, 3), Point(2, 3)])
Out[64]:
   x  y
0  0  0
1  0  3
2  2  3
```

Missing data

Much more will be said on this topic in the *Missing data* section. To construct a DataFrame with missing data, we use `np.nan` to represent missing values. Alternatively, you may pass a `numpy.MaskedArray` as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

Alternate constructors

DataFrame.from_dict

`DataFrame.from_dict` takes a dict of dicts or a dict of array-like sequences and returns a DataFrame. It operates like the `DataFrame` constructor except for the `orient` parameter which is `'columns'` by default, but which can be set to `'index'` in order to use the dict keys as row labels.

```
In [65]: pd.DataFrame.from_dict(dict([("A", [1, 2, 3]), ("B", [4, 5, 6])]))
Out[65]:
   A  B
0  1  4
1  2  5
2  3  6
```

If you pass `orient='index'`, the keys will be the row labels. In this case, you can also pass the desired column names:

```
In [66]: pd.DataFrame.from_dict(
   ....:     dict([("A", [1, 2, 3]), ("B", [4, 5, 6])]),
   ....:     orient="index",
   ....:     columns=["one", "two", "three"],
   ....: )
   ....:
Out[66]:
   one  two  three
A    1    2      3
B    4    5      6
```

DataFrame.from_records

`DataFrame.from_records` takes a list of tuples or an ndarray with structured dtype. It works analogously to the normal `DataFrame` constructor, except that the resulting DataFrame index may be a specific field of the structured dtype. For example:

```
In [67]: data
Out[67]:
array([(1, 2., b'Hello'), (2, 3., b'World')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])

In [68]: pd.DataFrame.from_records(data, index="C")
Out[68]:
          A    B
C
b'Hello'  1  2.0
b'World'  2  3.0
```

Column selection, addition, deletion

You can treat a DataFrame semantically like a dict of like-indexed Series objects. Getting, setting, and deleting columns works with the same syntax as the analogous dict operations:

```
In [69]: df["one"]
Out[69]:
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64

In [70]: df["three"] = df["one"] * df["two"]

In [71]: df["flag"] = df["one"] > 2

In [72]: df
Out[72]:
   one  two  three   flag
a  1.0  1.0    1.0  False
b  2.0  2.0    4.0  False
c  3.0  3.0    9.0   True
d  NaN  4.0    NaN  False
```

Columns can be deleted or popped like with a dict:

```
In [73]: del df["two"]

In [74]: three = df.pop("three")

In [75]: df
Out[75]:
   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
```

When inserting a scalar value, it will naturally be propagated to fill the column:

```
In [76]: df["foo"] = "bar"

In [77]: df
Out[77]:
   one   flag  foo
a  1.0  False  bar
b  2.0  False  bar
c  3.0   True  bar
d  NaN  False  bar
```

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame's index:

```
In [78]: df["one_trunc"] = df["one"][:2]

In [79]: df
Out[79]:
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
```

You can insert raw ndarrays but their length must match the length of the DataFrame's index.

By default, columns get inserted at the end. The `insert` function is available to insert at a particular location in the columns:

```
In [80]: df.insert(1, "bar", df["one"])

In [81]: df
Out[81]:
   one  bar   flag  foo  one_trunc
a  1.0  1.0  False  bar        1.0
b  2.0  2.0  False  bar        2.0
c  3.0  3.0   True  bar        NaN
d  NaN  NaN  False  bar        NaN
```

Assigning new columns in method chains

Inspired by dplyr's `mutate` verb, DataFrame has an *assign()* method that allows you to easily create new columns that are potentially derived from existing columns.

```
In [82]: iris = pd.read_csv("data/iris.data")

In [83]: iris.head()
Out[83]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
3          4.6         3.1          1.5         0.2  Iris-setosa
4          5.0         3.6          1.4         0.2  Iris-setosa

In [84]: iris.assign(sepal_ratio=iris["SepalWidth"] / iris["SepalLength"]).head()
Out[84]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

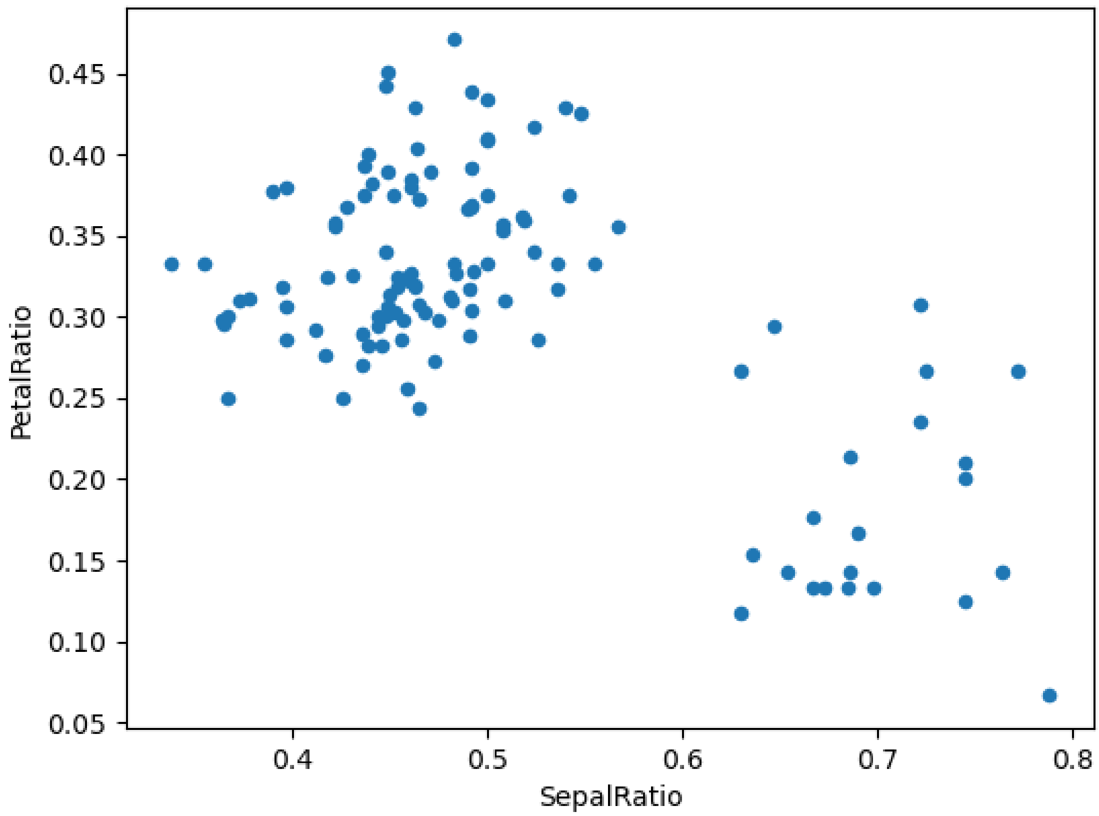
In the example above, we inserted a precomputed value. We can also pass in a function of one argument to be evaluated on the DataFrame being assigned to.

```
In [85]: iris.assign(sepal_ratio=lambda x: (x["SepalWidth"] / x["SepalLength"])).head()
Out[85]:
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
```

`assign` **always** returns a copy of the data, leaving the original DataFrame untouched.

Passing a callable, as opposed to an actual value to be inserted, is useful when you don't have a reference to the DataFrame at hand. This is common when using `assign` in a chain of operations. For example, we can limit the DataFrame to just those observations with a Sepal Length greater than 5, calculate the ratio, and plot:

```
In [86]: (
   ....:     iris.query("SepalLength > 5")
   ....:     .assign(
   ....:         SepalRatio=lambda x: x.SepalWidth / x.SepalLength,
   ....:         PetalRatio=lambda x: x.PetalWidth / x.PetalLength,
   ....:     )
   ....:     .plot(kind="scatter", x="SepalRatio", y="PetalRatio")
   ....: )
   ....:
Out[86]: <AxesSubplot:xlabel='SepalRatio', ylabel='PetalRatio'>
```



Since a function is passed in, the function is computed on the DataFrame being assigned to. Importantly, this is the DataFrame that's been filtered to those rows with sepal length greater than 5. The filtering happens first, and then the ratio calculations. This is an example where we didn't have a reference to the *filtered* DataFrame available.

The function signature for assign is simply `**kwargs`. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a `Series` or NumPy array), or a function of one argument to be called on the `DataFrame`. A *copy* of the original DataFrame is returned, with the new values inserted.

Starting with Python 3.6 the order of `**kwargs` is preserved. This allows for *dependent* assignment, where an expression later in `**kwargs` can refer to a column created earlier in the same `assign()`.

```
In [87]: dfa = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})

In [88]: dfa.assign(C=lambda x: x["A"] + x["B"], D=lambda x: x["A"] + x["C"])
Out[88]:
   A  B  C   D
0  1  4  5   6
1  2  5  7   9
2  3  6  9  12
```

In the second expression, `x['C']` will refer to the newly created column, that's equal to `dfa['A'] + dfa['B']`.

Indexing / selection

The basics of indexing are as follows:

Operation	Syntax	Result
Select column	df[col]	Series
Select row by label	df.loc[label]	Series
Select row by integer location	df.iloc[loc]	Series
Slice rows	df[5:10]	DataFrame
Select rows by boolean vector	df[bool_vec]	DataFrame

Row selection, for example, returns a Series whose index is the columns of the DataFrame:

```
In [89]: df.loc["b"]
Out[89]:
one              2.0
bar              2.0
flag           False
foo              bar
one_trunc        2.0
Name: b, dtype: object

In [90]: df.iloc[2]
Out[90]:
one             3.0
bar             3.0
flag           True
foo             bar
one_trunc       NaN
Name: c, dtype: object
```

For a more exhaustive treatment of sophisticated label-based indexing and slicing, see the *section on indexing*. We will address the fundamentals of reindexing / conforming to new sets of labels in the *section on reindexing*.

Data alignment and arithmetic

Data alignment between DataFrame objects automatically align on **both the columns and the index (row labels)**. Again, the resulting object will have the union of the column and row labels.

```
In [91]: df = pd.DataFrame(np.random.randn(10, 4), columns=["A", "B", "C", "D"])

In [92]: df2 = pd.DataFrame(np.random.randn(7, 3), columns=["A", "B", "C"])

In [93]: df + df2
Out[93]:
          A         B         C   D
0  0.045691 -0.014138  1.380871 NaN
1 -0.955398 -1.501007  0.037181 NaN
2 -0.662690  1.534833 -0.859691 NaN
3 -2.452949  1.237274 -0.133712 NaN
4  1.414490  1.951676 -2.320422 NaN
5 -0.494922 -1.649727 -1.084601 NaN
```

(continues on next page)

(continued from previous page)

```
6 -1.047551 -0.748572 -0.805479 NaN
7       NaN       NaN       NaN NaN
8       NaN       NaN       NaN NaN
9       NaN       NaN       NaN NaN
```

When doing an operation between DataFrame and Series, the default behavior is to align the Series **index** on the DataFrame **columns**, thus broadcasting row-wise. For example:

```
In [94]: df - df.iloc[0]
Out[94]:
          A         B         C         D
0  0.000000  0.000000  0.000000  0.000000
1 -1.359261 -0.248717 -0.453372 -1.754659
2  0.253128  0.829678  0.010026 -1.991234
3 -1.311128  0.054325 -1.724913 -1.620544
4  0.573025  1.500742 -0.676070  1.367331
5 -1.741248  0.781993 -1.241620 -2.053136
6 -1.240774 -0.869551 -0.153282  0.000430
7 -0.743894  0.411013 -0.929563 -0.282386
8 -1.194921  1.320690  0.238224 -1.482644
9  2.293786  1.856228  0.773289 -1.446531
```

For explicit control over the matching and broadcasting behavior, see the section on *flexible binary operations*.

Operations with scalars are just as you would expect:

```
In [95]: df * 5 + 2
Out[95]:
           A         B         C          D
0   3.359299 -0.124862  4.835102   3.381160
1  -3.437003 -1.368449  2.568242  -5.392133
2   4.624938  4.023526  4.885230  -6.575010
3  -3.196342  0.146766 -3.789461  -4.721559
4   6.224426  7.378849  1.454750  10.217815
5  -5.346940  3.785103 -1.373001  -6.884519
6  -2.844569 -4.472618  4.068691   3.383309
7  -0.360173  1.930201  0.187285   1.969232
8  -2.615303  6.478587  6.026220  -4.032059
9  14.828230  9.156280  8.701544  -3.851494

In [96]: 1 / df
Out[96]:
          A          B         C           D
0  3.678365  -2.353094  1.763605    3.620145
1 -0.919624  -1.484363  8.799067   -0.676395
2  1.904807   2.470934  1.732964   -0.583090
3 -0.962215  -2.697986 -0.863638   -0.743875
4  1.183593   0.929567 -9.170108    0.608434
5 -0.680555   2.800959 -1.482360   -0.562777
6 -1.032084  -0.772485  2.416988    3.614523
7 -2.118489 -71.634509 -2.758294 -162.507295
8 -1.083352   1.116424  1.241860   -0.828904
9  0.389765   0.698687  0.746097   -0.854483
```

(continues on next page)

(continued from previous page)

```
In [97]: df ** 4
Out[97]:
           A             B         C             D
0   0.005462  3.261689e-02  0.103370  5.822320e-03
1   1.398165  2.059869e-01  0.000167  4.777482e+00
2   0.075962  2.682596e-02  0.110877  8.650845e+00
3   1.166571  1.887302e-02  1.797515  3.265879e+00
4   0.509555  1.339298e+00  0.000141  7.297019e+00
5   4.661717  1.624699e-02  0.207103  9.969092e+00
6   0.881334  2.808277e+00  0.029302  5.858632e-03
7   0.049647  3.797614e-08  0.017276  1.433866e-09
8   0.725974  6.437005e-01  0.420446  2.118275e+00
9  43.329821  4.196326e+00  3.227153  1.875802e+00
```

Boolean operators work as well:

```
In [98]: df1 = pd.DataFrame({"a": [1, 0, 1], "b": [0, 1, 1]}, dtype=bool)

In [99]: df2 = pd.DataFrame({"a": [0, 1, 1], "b": [1, 1, 0]}, dtype=bool)

In [100]: df1 & df2
Out[100]:
       a      b
0  False  False
1  False   True
2   True  False

In [101]: df1 | df2
Out[101]:
      a     b
0  True  True
1  True  True
2  True  True

In [102]: df1 ^ df2
Out[102]:
       a      b
0   True   True
1   True  False
2  False   True

In [103]: -df1
Out[103]:
       a      b
0  False   True
1   True  False
2  False  False
```

Transposing

To transpose, access the `T` attribute (also the `transpose` function), similar to an ndarray:

```
# only show the first 5 rows
In [104]: df[:5].T
Out[104]:
          0         1         2         3         4
A  0.271860 -1.087401  0.524988 -1.039268  0.844885
B -0.424972 -0.673690  0.404705 -0.370647  1.075770
C  0.567020  0.113648  0.577046 -1.157892 -0.109050
D  0.276232 -1.478427 -1.715002 -1.344312  1.643563
```

DataFrame interoperability with NumPy functions

Elementwise NumPy ufuncs (log, exp, sqrt, ...) and various other NumPy functions can be used with no issues on Series and DataFrame, assuming the data within are numeric:

```
In [105]: np.exp(df)
Out[105]:
           A         B         C         D
0   1.312403  0.653788  1.763006  1.318154
1   0.337092  0.509824  1.120358  0.227996
2   1.690438  1.498861  1.780770  0.179963
3   0.353713  0.690288  0.314148  0.260719
4   2.327710  2.932249  0.896686  5.173571
5   0.230066  1.429065  0.509360  0.169161
6   0.379495  0.274028  1.512461  1.318720
7   0.623732  0.986137  0.695904  0.993865
8   0.397301  2.449092  2.237242  0.299269
9  13.009059  4.183951  3.820223  0.310274

In [106]: np.asarray(df)
Out[106]:
array([[ 0.2719, -0.425 ,  0.567 ,  0.2762],
       [-1.0874, -0.6737,  0.1136, -1.4784],
       [ 0.525 ,  0.4047,  0.577 , -1.715 ],
       [-1.0393, -0.3706, -1.1579, -1.3443],
       [ 0.8449,  1.0758, -0.109 ,  1.6436],
       [-1.4694,  0.357 , -0.6746, -1.7769],
       [-0.9689, -1.2945,  0.4137,  0.2767],
       [-0.472 , -0.014 , -0.3625, -0.0062],
       [-0.9231,  0.8957,  0.8052, -1.2064],
       [ 2.5656,  1.4313,  1.3403, -1.1703]])
```

DataFrame is not intended to be a drop-in replacement for ndarray as its indexing semantics and data model are quite different in places from an n-dimensional array.

`Series` implements `__array_ufunc__`, which allows it to work with NumPy's universal functions.

The ufunc is applied to the underlying array in a Series.

```
In [107]: ser = pd.Series([1, 2, 3, 4])
```

(continues on next page)

(continued from previous page)

```
In [108]: np.exp(ser)
Out[108]:
0     2.718282
1     7.389056
2    20.085537
3    54.598150
dtype: float64
```

Changed in version 0.25.0: When multiple `Series` are passed to a ufunc, they are aligned before performing the operation.

Like other parts of the library, pandas will automatically align labeled inputs as part of a ufunc with multiple inputs. For example, using `numpy.remainder()` on two `Series` with differently ordered labels will align before the operation.

```
In [109]: ser1 = pd.Series([1, 2, 3], index=["a", "b", "c"])

In [110]: ser2 = pd.Series([1, 3, 5], index=["b", "a", "c"])

In [111]: ser1
Out[111]:
a    1
b    2
c    3
dtype: int64

In [112]: ser2
Out[112]:
b    1
a    3
c    5
dtype: int64

In [113]: np.remainder(ser1, ser2)
Out[113]:
a    1
b    0
c    3
dtype: int64
```

As usual, the union of the two indices is taken, and non-overlapping values are filled with missing values.

```
In [114]: ser3 = pd.Series([2, 4, 6], index=["b", "c", "d"])

In [115]: ser3
Out[115]:
b    2
c    4
d    6
dtype: int64

In [116]: np.remainder(ser1, ser3)
Out[116]:
a    NaN
```

(continues on next page)

(continued from previous page)

```
b    0.0
c    3.0
d    NaN
dtype: float64
```

When a binary ufunc is applied to a *Series* and *Index*, the Series implementation takes precedence and a Series is returned.

```
In [117]: ser = pd.Series([1, 2, 3])

In [118]: idx = pd.Index([4, 5, 6])

In [119]: np.maximum(ser, idx)
Out[119]:
0    4
1    5
2    6
dtype: int64
```

NumPy ufuncs are safe to apply to *Series* backed by non-ndarray arrays, for example *arrays.SparseArray* (see *Sparse calculation*). If possible, the ufunc is applied without converting the underlying data to an ndarray.

Console display

Very large DataFrames will be truncated to display them in the console. You can also get a summary using *info()*. (Here I am reading a CSV version of the **baseball** dataset from the **plyr** R package):

```
In [120]: baseball = pd.read_csv("data/baseball.csv")

In [121]: print(baseball)
       id     player  year  stint team  lg   g   ab   r    h  X2b  X3b  hr   rbi   sb  ␣
↪cs  bb    so  ibb  hbp   sh   sf  gidp
0   88641  womacto01  2006      2  CHN  NL  19   50   6   14    1    0   1   2.0  1.0  1.
↪0   4   4.0  0.0  0.0  3.0  0.0   0.0
1   88643  schilcu01  2006      1  BOS  AL  31    2   0    1    0    0   0   0.0  0.0  0.
↪0   0   1.0  0.0  0.0  0.0  0.0   0.0
..    ...        ...   ...    ...  ...  ..  ..  ...  ..  ...  ...  ...  ..   ...  ...  ..
↪.  ..   ...  ...  ...  ...  ...   ...
98  89533   aloumo01  2007      1  NYN  NL  87  328  51  112   19    1  13  49.0  3.0  0.
↪0  27  30.0  5.0  2.0  0.0  3.0  13.0
99  89534  alomasa02  2007      1  NYN  NL   8   22   1    3    1    0   0   0.0  0.0  0.
↪0   0   3.0  0.0  0.0  0.0  0.0   0.0

[100 rows x 23 columns]

In [122]: baseball.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 23 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   id      100 non-null    int64
```

(continues on next page)

(continued from previous page)

```
 1   player  100 non-null    object
 2   year    100 non-null    int64
 3   stint   100 non-null    int64
 4   team    100 non-null    object
 5   lg      100 non-null    object
 6   g       100 non-null    int64
 7   ab      100 non-null    int64
 8   r       100 non-null    int64
 9   h       100 non-null    int64
 10  X2b     100 non-null    int64
 11  X3b     100 non-null    int64
 12  hr      100 non-null    int64
 13  rbi     100 non-null    float64
 14  sb      100 non-null    float64
 15  cs      100 non-null    float64
 16  bb      100 non-null    int64
 17  so      100 non-null    float64
 18  ibb     100 non-null    float64
 19  hbp     100 non-null    float64
 20  sh      100 non-null    float64
 21  sf      100 non-null    float64
 22  gidp    100 non-null    float64
dtypes: float64(9), int64(11), object(3)
memory usage: 18.1+ KB
```

However, using `to_string` will return a string representation of the DataFrame in tabular form, though it won't always fit the console width:

```
In [123]: print(baseball.iloc[-20:, :12].to_string())
       id     player  year  stint team  lg    g   ab   r    h  X2b  X3b
80  89474  finlest01  2007      1  COL  NL   43   94   9   17    3    0
81  89480  embreal01  2007      1  OAK  AL    4    0   0    0    0    0
82  89481  edmonji01  2007      1  SLN  NL  117  365  39   92   15    2
83  89482  easleda01  2007      1  NYN  NL   76  193  24   54    6    0
84  89489  delgaca01  2007      1  NYN  NL  139  538  71  139   30    0
85  89493  cormirh01  2007      1  CIN  NL    6    0   0    0    0    0
86  89494  coninje01  2007      2  NYN  NL   21   41   2    8    2    0
87  89495  coninje01  2007      1  CIN  NL   80  215  23   57   11    1
88  89497  clemero02  2007      1  NYA  AL    2    2   0    1    0    0
89  89498  claytro01  2007      2  BOS  AL    8    6   1    0    0    0
90  89499  claytro01  2007      1  TOR  AL   69  189  23   48   14    0
91  89501  cirilje01  2007      2  ARI  NL   28   40   6    8    4    0
92  89502  cirilje01  2007      1  MIN  AL   50  153  18   40    9    2
93  89521  bondsba01  2007      1  SFN  NL  126  340  75   94   14    0
94  89523  biggicr01  2007      1  HOU  NL  141  517  68  130   31    3
95  89525  benitar01  2007      2  FLO  NL   34    0   0    0    0    0
96  89526  benitar01  2007      1  SFN  NL   19    0   0    0    0    0
97  89530  ausmubr01  2007      1  HOU  NL  117  349  38   82   16    3
98  89533   aloumo01  2007      1  NYN  NL   87  328  51  112   19    1
99  89534  alomasa02  2007      1  NYN  NL    8   22   1    3    1    0
```

Wide DataFrames will be printed across multiple rows by default:

```
In [124]: pd.DataFrame(np.random.randn(3, 12))
Out[124]:
         0         1         2         3         4         5         6         7         ␣
↪8         9        10        11
0 -1.226825  0.769804 -1.281247 -0.727707 -0.121306 -0.097883  0.695775  0.341734  0.
↪959726 -1.110336 -0.619976  0.149748
1 -0.732339  0.687738  0.176444  0.403310 -0.154951  0.301624 -2.179861 -1.369849 -0.
↪954208  1.462696 -1.743161 -0.826591
2 -0.345352  1.314232  0.690579  0.995761  2.396780  0.014871  3.357427 -0.317441 -1.
↪236269  0.896171 -0.487602 -0.082240
```

You can change how much to print on a single row by setting the `display.width` option:

```
In [125]: pd.set_option("display.width", 40)  # default is 80

In [126]: pd.DataFrame(np.random.randn(3, 12))
Out[126]:
         0         1         2         3         4         5         6         7         ␣
↪8         9        10        11
0 -2.182937  0.380396  0.084844  0.432390  1.519970 -0.493662  0.600178  0.274230  0.
↪132885 -0.023688  2.410179  1.450520
1  0.206053 -0.251905 -2.213588  1.063327  1.266143  0.299368 -0.863838  0.408204 -1.
↪048089 -0.025747 -0.988387  0.094055
2  1.262731  1.289997  0.082423 -0.055758  0.536580 -0.489682  0.369374 -0.034571 -2.
↪484478 -0.281461  0.030711  0.109121
```

You can adjust the max width of the individual columns by setting `display.max_colwidth`

```
In [127]: datafile = {
   .....:     "filename": ["filename_01", "filename_02"],
   .....:     "path": [
   .....:         "media/user_name/storage/folder_01/filename_01",
   .....:         "media/user_name/storage/folder_02/filename_02",
   .....:     ],
   .....: }
   .....:

In [128]: pd.set_option("display.max_colwidth", 30)

In [129]: pd.DataFrame(datafile)
Out[129]:
      filename                           path
0  filename_01  media/user_name/storage/fo...
1  filename_02  media/user_name/storage/fo...

In [130]: pd.set_option("display.max_colwidth", 100)

In [131]: pd.DataFrame(datafile)
Out[131]:
      filename                                           path
0  filename_01  media/user_name/storage/folder_01/filename_01
1  filename_02  media/user_name/storage/folder_02/filename_02
```

You can also disable this feature via the `expand_frame_repr` option. This will print the table in one block.

DataFrame column attribute access and IPython completion

If a DataFrame column label is a valid Python variable name, the column can be accessed like an attribute:

```
In [132]: df = pd.DataFrame({"foo1": np.random.randn(5), "foo2": np.random.randn(5)})

In [133]: df
Out[133]:
       foo1      foo2
0  1.126203  0.781836
1 -0.977349 -1.071357
2  1.474071  0.441153
3 -0.064034  2.353925
4 -1.282782  0.583787

In [134]: df.foo1
Out[134]:
0    1.126203
1   -0.977349
2    1.474071
3   -0.064034
4   -1.282782
Name: foo1, dtype: float64
```

The columns are also connected to the IPython completion mechanism so they can be tab-completed:

```
In [5]: df.foo<TAB>  # noqa: E225, E999
df.foo1  df.foo2
```

2.3 Essential basic functionality

Here we discuss a lot of the essential functionality common to the pandas data structures. To begin, let's create some example objects like we did in the *10 minutes to pandas* section:

```
In [1]: index = pd.date_range("1/1/2000", periods=8)

In [2]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])

In [3]: df = pd.DataFrame(np.random.randn(8, 3), index=index, columns=["A", "B", "C"])
```

2.3.1 Head and tail

To view a small sample of a Series or DataFrame object, use the `head()` and `tail()` methods. The default number of elements to display is five, but you may pass a custom number.

```
In [4]: long_series = pd.Series(np.random.randn(1000))

In [5]: long_series.head()
Out[5]:
0   -1.157892
```

(continues on next page)

(continued from previous page)

```
1   -1.344312
2    0.844885
3    1.075770
4   -0.109050
dtype: float64

In [6]: long_series.tail(3)
Out[6]:
997   -0.289388
998   -1.020544
999    0.589993
dtype: float64
```

2.3.2 Attributes and underlying data

pandas objects have a number of attributes enabling you to access the metadata

- **shape**: gives the axis dimensions of the object, consistent with ndarray
- **Axis labels**
  - **Series**: *index* (only axis)
  - **DataFrame**: *index* (rows) and *columns*

Note, **these attributes can be safely assigned to**!

```
In [7]: df[:2]
Out[7]:
                   A         B         C
2000-01-01 -0.173215  0.119209 -1.044236
2000-01-02 -0.861849 -2.104569 -0.494929

In [8]: df.columns = [x.lower() for x in df.columns]

In [9]: df
Out[9]:
                   a         b         c
2000-01-01 -0.173215  0.119209 -1.044236
2000-01-02 -0.861849 -2.104569 -0.494929
2000-01-03  1.071804  0.721555 -0.706771
2000-01-04 -1.039575  0.271860 -0.424972
2000-01-05  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427
2000-01-07  0.524988  0.404705  0.577046
2000-01-08 -1.715002 -1.039268 -0.370647
```

pandas objects (`Index`, `Series`, `DataFrame`) can be thought of as containers for arrays, which hold the actual data and do the actual computation. For many types, the underlying array is a `numpy.ndarray`. However, pandas and 3rd party libraries may *extend* NumPy's type system to add support for custom arrays (see *dtypes*).

To get the actual data inside a `Index` or `Series`, use the `.array` property

```
In [10]: s.array
Out[10]:
<PandasArray>
[ 0.4691122999071863, -0.2828633443286633, -1.5090585031735124,
 -1.1356323710171934,  1.2121120250208506]
Length: 5, dtype: float64

In [11]: s.index.array
Out[11]:
<PandasArray>
['a', 'b', 'c', 'd', 'e']
Length: 5, dtype: object
```

`array` will always be an `ExtensionArray`. The exact details of what an `ExtensionArray` is and why pandas uses them are a bit beyond the scope of this introduction. See *dtypes* for more.

If you know you need a NumPy array, use `to_numpy()` or `numpy.asarray()`.

```
In [12]: s.to_numpy()
Out[12]: array([ 0.4691, -0.2829, -1.5091, -1.1356,  1.2121])

In [13]: np.asarray(s)
Out[13]: array([ 0.4691, -0.2829, -1.5091, -1.1356,  1.2121])
```

When the Series or Index is backed by an `ExtensionArray`, `to_numpy()` may involve copying data and coercing values. See *dtypes* for more.

`to_numpy()` gives some control over the `dtype` of the resulting `numpy.ndarray`. For example, consider datetimes with timezones. NumPy doesn't have a dtype to represent timezone-aware datetimes, so there are two possibly useful representations:

1. An object-dtype `numpy.ndarray` with `Timestamp` objects, each with the correct `tz`
2. A `datetime64[ns]` -dtype `numpy.ndarray`, where the values have been converted to UTC and the timezone discarded

Timezones may be preserved with `dtype=object`

```
In [14]: ser = pd.Series(pd.date_range("2000", periods=2, tz="CET"))

In [15]: ser.to_numpy(dtype=object)
Out[15]:
array([Timestamp('2000-01-01 00:00:00+0100', tz='CET'),
       Timestamp('2000-01-02 00:00:00+0100', tz='CET')], dtype=object)
```

Or thrown away with `dtype='datetime64[ns]'`

```
In [16]: ser.to_numpy(dtype="datetime64[ns]")
Out[16]:
array(['1999-12-31T23:00:00.000000000', '2000-01-01T23:00:00.000000000'],
      dtype='datetime64[ns]')
```

Getting the "raw data" inside a `DataFrame` is possibly a bit more complex. When your `DataFrame` only has a single data type for all the columns, `DataFrame.to_numpy()` will return the underlying data:

```
In [17]: df.to_numpy()
Out[17]:
array([[-0.1732,  0.1192, -1.0442],
       [-0.8618, -2.1046, -0.4949],
       [ 1.0718,  0.7216, -0.7068],
       [-1.0396,  0.2719, -0.425 ],
       [ 0.567 ,  0.2762, -1.0874],
       [-0.6737,  0.1136, -1.4784],
       [ 0.525 ,  0.4047,  0.577 ],
       [-1.715 , -1.0393, -0.3706]])
```

If a DataFrame contains homogeneously-typed data, the ndarray can actually be modified in-place, and the changes will be reflected in the data structure. For heterogeneous data (e.g. some of the DataFrame's columns are not all the same dtype), this will not be the case. The values attribute itself, unlike the axis labels, cannot be assigned to.

Note: When working with heterogeneous data, the dtype of the resulting ndarray will be chosen to accommodate all of the data involved. For example, if strings are involved, the result will be of object dtype. If there are only floats and integers, the resulting array will be of float dtype.

In the past, pandas recommended *Series.values* or *DataFrame.values* for extracting the data from a Series or DataFrame. You'll still find references to these in old code bases and online. Going forward, we recommend avoiding `.values` and using `.array` or `.to_numpy()`. `.values` has the following drawbacks:

1. When your Series contains an *extension type*, it's unclear whether *Series.values* returns a NumPy array or the extension array. *Series.array* will always return an *ExtensionArray*, and will never copy data. *Series.to_numpy()* will always return a NumPy array, potentially at the cost of copying / coercing values.
2. When your DataFrame contains a mixture of data types, *DataFrame.values* may involve copying data and coercing values to a common dtype, a relatively expensive operation. *DataFrame.to_numpy()*, being a method, makes it clearer that the returned NumPy array may not be a view on the same data in the DataFrame.

2.3.3 Accelerated operations

pandas has support for accelerating certain types of binary numerical and boolean operations using the `numexpr` library and the `bottleneck` libraries.

These libraries are especially useful when dealing with large data sets, and provide large speedups. `numexpr` uses smart chunking, caching, and multiple cores. `bottleneck` is a set of specialized cython routines that are especially fast when dealing with arrays that have `nans`.

Here is a sample (using 100 column x 100,000 row `DataFrames`):

Operation	0.11.0 (ms)	Prior Version (ms)	Ratio to Prior
`df1 > df2`	13.32	125.35	0.1063
`df1 * df2`	21.71	36.63	0.5928
`df1 + df2`	22.04	36.50	0.6039

You are highly encouraged to install both libraries. See the section *Recommended Dependencies* for more installation info.

These are both enabled to be used by default, you can control this by setting the options:

```
pd.set_option("compute.use_bottleneck", False)
pd.set_option("compute.use_numexpr", False)
```

2.3.4 Flexible binary operations

With binary operations between pandas data structures, there are two key points of interest:

- Broadcasting behavior between higher- (e.g. DataFrame) and lower-dimensional (e.g. Series) objects.
- Missing data in computations.

We will demonstrate how to manage these issues independently, though they can be handled simultaneously.

Matching / broadcasting behavior

DataFrame has the methods `add()`, `sub()`, `mul()`, `div()` and related functions `radd()`, `rsub()`, . . . for carrying out binary operations. For broadcasting behavior, Series input is of primary interest. Using these functions, you can use to either match on the *index* or *columns* via the **axis** keyword:

```
In [18]: df = pd.DataFrame(
   ....:     {
   ....:         "one": pd.Series(np.random.randn(3), index=["a", "b", "c"]),
   ....:         "two": pd.Series(np.random.randn(4), index=["a", "b", "c", "d"]),
   ....:         "three": pd.Series(np.random.randn(3), index=["b", "c", "d"]),
   ....:     }
   ....: )
   ....:

In [19]: df
Out[19]:
        one       two     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [20]: row = df.iloc[1]

In [21]: column = df["two"]

In [22]: df.sub(row, axis="columns")
Out[22]:
        one       two     three
a  1.051928 -0.139606       NaN
b  0.000000  0.000000  0.000000
c  0.352192 -0.433754  1.277825
d       NaN -1.632779 -0.562782

In [23]: df.sub(row, axis=1)
Out[23]:
        one       two     three
a  1.051928 -0.139606       NaN
```

(continues on next page)

(continued from previous page)

```
b  0.000000  0.000000  0.000000
c  0.352192 -0.433754  1.277825
d       NaN -1.632779 -0.562782

In [24]: df.sub(column, axis="index")
Out[24]:
        one  two     three
a -0.377535  0.0       NaN
b -1.569069  0.0 -1.962513
c -0.783123  0.0 -0.250933
d       NaN  0.0 -0.892516

In [25]: df.sub(column, axis=0)
Out[25]:
        one  two     three
a -0.377535  0.0       NaN
b -1.569069  0.0 -1.962513
c -0.783123  0.0 -0.250933
d       NaN  0.0 -0.892516
```

Furthermore you can align a level of a MultiIndexed DataFrame with a Series.

```
In [26]: dfmi = df.copy()

In [27]: dfmi.index = pd.MultiIndex.from_tuples(
   ....:     [(1, "a"), (1, "b"), (1, "c"), (2, "a")], names=["first", "second"]
   ....: )
   ....:

In [28]: dfmi.sub(column, axis=0, level="second")
Out[28]:
                   one       two     three
first second
1     a      -0.377535  0.000000       NaN
      b      -1.569069  0.000000 -1.962513
      c      -0.783123  0.000000 -0.250933
2     a            NaN -1.493173 -2.385688
```

Series and Index also support the `divmod()` builtin. This function takes the floor division and modulo operation at the same time returning a two-tuple of the same type as the left hand side. For example:

```
In [29]: s = pd.Series(np.arange(10))

In [30]: s
Out[30]:
0    0
1    1
2    2
3    3
4    4
5    5
6    6
```

(continues on next page)

(continued from previous page)

```
7    7
8    8
9    9
dtype: int64

In [31]: div, rem = divmod(s, 3)

In [32]: div
Out[32]:
0    0
1    0
2    0
3    1
4    1
5    1
6    2
7    2
8    2
9    3
dtype: int64

In [33]: rem
Out[33]:
0    0
1    1
2    2
3    0
4    1
5    2
6    0
7    1
8    2
9    0
dtype: int64

In [34]: idx = pd.Index(np.arange(10))

In [35]: idx
Out[35]: Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')

In [36]: div, rem = divmod(idx, 3)

In [37]: div
Out[37]: Int64Index([0, 0, 0, 1, 1, 1, 2, 2, 2, 3], dtype='int64')

In [38]: rem
Out[38]: Int64Index([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype='int64')
```

We can also do elementwise divmod():

```
In [39]: div, rem = divmod(s, [2, 2, 3, 3, 4, 4, 5, 5, 6, 6])
```

(continues on next page)

(continued from previous page)

```
In [40]: div
Out[40]:
0    0
1    0
2    0
3    1
4    1
5    1
6    1
7    1
8    1
9    1
dtype: int64

In [41]: rem
Out[41]:
0    0
1    1
2    2
3    0
4    0
5    1
6    1
7    2
8    2
9    3
dtype: int64
```

Missing data / operations with fill values

In Series and DataFrame, the arithmetic functions have the option of inputting a *fill_value*, namely a value to substitute when at most one of the values at a location are missing. For example, when adding two DataFrame objects, you may wish to treat NaN as 0 unless both DataFrames are missing that value, in which case the result will be NaN (you can later replace NaN with some other value using `fillna` if you wish).

```
In [42]: df
Out[42]:
        one       two     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [43]: df2
Out[43]:
        one       two     three
a  1.394981  1.772517  1.000000
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172
```

(continues on next page)

(continued from previous page)

```
In [44]: df + df2
Out[44]:
        one       two     three
a  2.789963  3.545034       NaN
b  0.686107  3.824246 -0.100780
c  1.390491  2.956737  2.454870
d       NaN  0.558688 -1.226343

In [45]: df.add(df2, fill_value=0)
Out[45]:
        one       two     three
a  2.789963  3.545034  1.000000
b  0.686107  3.824246 -0.100780
c  1.390491  2.956737  2.454870
d       NaN  0.558688 -1.226343
```

Flexible comparisons

Series and DataFrame have the binary comparison methods `eq`, `ne`, `lt`, `gt`, `le`, and `ge` whose behavior is analogous to the binary arithmetic operations described above:

```
In [46]: df.gt(df2)
Out[46]:
     one    two  three
a  False  False  False
b  False  False  False
c  False  False  False
d  False  False  False

In [47]: df2.ne(df)
Out[47]:
     one    two  three
a  False  False   True
b  False  False  False
c  False  False  False
d   True  False  False
```

These operations produce a pandas object of the same type as the left-hand-side input that is of dtype `bool`. These `boolean` objects can be used in indexing operations, see the section on *Boolean indexing*.

Boolean reductions

You can apply the reductions: *`empty`*, *`any()`*, *`all()`*, and *`bool()`* to provide a way to summarize a boolean result.

```
In [48]: (df > 0).all()
Out[48]:
one      False
two       True
three    False
dtype: bool
```

(continues on next page)

(continued from previous page)

```
In [49]: (df > 0).any()
Out[49]:
one      True
two      True
three    True
dtype: bool
```

You can reduce to a final boolean value.

```
In [50]: (df > 0).any().any()
Out[50]: True
```

You can test if a pandas object is empty, via the *empty* property.

```
In [51]: df.empty
Out[51]: False

In [52]: pd.DataFrame(columns=list("ABC")).empty
Out[52]: True
```

To evaluate single-element pandas objects in a boolean context, use the method *bool()*:

```
In [53]: pd.Series([True]).bool()
Out[53]: True

In [54]: pd.Series([False]).bool()
Out[54]: False

In [55]: pd.DataFrame([[True]]).bool()
Out[55]: True

In [56]: pd.DataFrame([[False]]).bool()
Out[56]: False
```

Warning: You might be tempted to do the following:

```
>>> if df:
...     pass
```

Or

```
>>> df and df2
```

These will both raise errors, as you are trying to compare multiple values.:

```
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

See *gotchas* for a more detailed discussion.

Comparing if objects are equivalent

Often you may find that there is more than one way to compute the same result. As a simple example, consider `df + df` and `df * 2`. To test that these two computations produce the same result, given the tools shown above, you might imagine using `(df + df == df * 2).all()`. But in fact, this expression is False:

```
In [57]: df + df == df * 2
Out[57]:
     one   two  three
a   True  True  False
b   True  True   True
c   True  True   True
d  False  True   True

In [58]: (df + df == df * 2).all()
Out[58]:
one      False
two       True
three    False
dtype: bool
```

Notice that the boolean DataFrame `df + df == df * 2` contains some False values! This is because NaNs do not compare as equals:

```
In [59]: np.nan == np.nan
Out[59]: False
```

So, NDFrames (such as Series and DataFrames) have an *equals()* method for testing equality, with NaNs in corresponding locations treated as equal.

```
In [60]: (df + df).equals(df * 2)
Out[60]: True
```

Note that the Series or DataFrame index needs to be in the same order for equality to be True:

```
In [61]: df1 = pd.DataFrame({"col": ["foo", 0, np.nan]})

In [62]: df2 = pd.DataFrame({"col": [np.nan, 0, "foo"]}, index=[2, 1, 0])

In [63]: df1.equals(df2)
Out[63]: False

In [64]: df1.equals(df2.sort_index())
Out[64]: True
```

Comparing array-like objects

You can conveniently perform element-wise comparisons when comparing a pandas data structure with a scalar value:

```
In [65]: pd.Series(["foo", "bar", "baz"]) == "foo"
Out[65]:
0     True
1    False
2    False
dtype: bool

In [66]: pd.Index(["foo", "bar", "baz"]) == "foo"
Out[66]: array([ True, False, False])
```

pandas also handles element-wise comparisons between different array-like objects of the same length:

```
In [67]: pd.Series(["foo", "bar", "baz"]) == pd.Index(["foo", "bar", "qux"])
Out[67]:
0     True
1     True
2    False
dtype: bool

In [68]: pd.Series(["foo", "bar", "baz"]) == np.array(["foo", "bar", "qux"])
Out[68]:
0     True
1     True
2    False
dtype: bool
```

Trying to compare `Index` or `Series` objects of different lengths will raise a ValueError:

```
In [55]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo', 'bar'])
ValueError: Series lengths must match to compare

In [56]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo'])
ValueError: Series lengths must match to compare
```

Note that this is different from the NumPy behavior where a comparison can be broadcast:

```
In [69]: np.array([1, 2, 3]) == np.array([2])
Out[69]: array([False,  True, False])
```

or it can return False if broadcasting can not be done:

```
In [70]: np.array([1, 2, 3]) == np.array([1, 2])
Out[70]: False
```

Combining overlapping data sets

A problem occasionally arising is the combination of two similar data sets where values in one are preferred over the other. An example would be two data series representing a particular economic indicator where one is considered to be of "higher quality". However, the lower quality series might extend further back in history or have more complete data coverage. As such, we would like to combine two DataFrame objects where missing values in one DataFrame are conditionally filled with like-labeled values from the other DataFrame. The function implementing this operation is *combine_first()*, which we illustrate:

```
In [71]: df1 = pd.DataFrame(
   ....:     {"A": [1.0, np.nan, 3.0, 5.0, np.nan], "B": [np.nan, 2.0, 3.0, np.nan, 6.0]}
   ....: )
   ....:

In [72]: df2 = pd.DataFrame(
   ....:     {
   ....:         "A": [5.0, 2.0, 4.0, np.nan, 3.0, 7.0],
   ....:         "B": [np.nan, np.nan, 3.0, 4.0, 6.0, 8.0],
   ....:     }
   ....: )
   ....:

In [73]: df1
Out[73]:
     A    B
0  1.0  NaN
1  NaN  2.0
2  3.0  3.0
3  5.0  NaN
4  NaN  6.0

In [74]: df2
Out[74]:
     A    B
0  5.0  NaN
1  2.0  NaN
2  4.0  3.0
3  NaN  4.0
4  3.0  6.0
5  7.0  8.0

In [75]: df1.combine_first(df2)
Out[75]:
     A    B
0  1.0  NaN
1  2.0  2.0
2  3.0  3.0
3  5.0  4.0
4  3.0  6.0
5  7.0  8.0
```

General DataFrame combine

The `combine_first()` method above calls the more general `DataFrame.combine()`. This method takes another DataFrame and a combiner function, aligns the input DataFrame and then passes the combiner function pairs of Series (i.e., columns whose names are the same).

So, for instance, to reproduce `combine_first()` as above:

```
In [76]: def combiner(x, y):
   ....:     return np.where(pd.isna(x), y, x)
   ....:

In [77]: df1.combine(df2, combiner)
Out[77]:
     A    B
0  1.0  NaN
1  2.0  2.0
2  3.0  3.0
3  5.0  4.0
4  3.0  6.0
5  7.0  8.0
```

2.3.5 Descriptive statistics

There exists a large number of methods for computing descriptive statistics and other related operations on *Series*, *DataFrame*. Most of these are aggregations (hence producing a lower-dimensional result) like `sum()`, `mean()`, and `quantile()`, but some of them, like `cumsum()` and `cumprod()`, produce an object of the same size. Generally speaking, these methods take an **axis** argument, just like *ndarray.{sum, std, …}*, but the axis can be specified by name or integer:

- **Series**: no axis argument needed
- **DataFrame**: "index" (axis=0, default), "columns" (axis=1)

For example:

```
In [78]: df
Out[78]:
        one       two     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [79]: df.mean(0)
Out[79]:
one      0.811094
two      1.360588
three    0.187958
dtype: float64

In [80]: df.mean(1)
Out[80]:
a    1.583749
```

(continues on next page)

(continued from previous page)

```
b    0.734929
c    1.133683
d   -0.166914
dtype: float64
```

All such methods have a `skipna` option signaling whether to exclude missing data (`True` by default):

```
In [81]: df.sum(0, skipna=False)
Out[81]:
one           NaN
two      5.442353
three         NaN
dtype: float64

In [82]: df.sum(axis=1, skipna=True)
Out[82]:
a    3.167498
b    2.204786
c    3.401050
d   -0.333828
dtype: float64
```

Combined with the broadcasting / arithmetic behavior, one can describe various statistical procedures, like standardization (rendering data zero mean and standard deviation of 1), very concisely:

```
In [83]: ts_stand = (df - df.mean()) / df.std()

In [84]: ts_stand.std()
Out[84]:
one      1.0
two      1.0
three    1.0
dtype: float64

In [85]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)

In [86]: xs_stand.std(1)
Out[86]:
a    1.0
b    1.0
c    1.0
d    1.0
dtype: float64
```

Note that methods like *cumsum()* and *cumprod()* preserve the location of `NaN` values. This is somewhat different from *expanding()* and *rolling()* since `NaN` behavior is furthermore dictated by a `min_periods` parameter.

```
In [87]: df.cumsum()
Out[87]:
        one       two     three
a  1.394981  1.772517       NaN
b  1.738035  3.684640 -0.050390
```

(continues on next page)

(continued from previous page)

```
c  2.433281  5.163008  1.177045
d       NaN  5.442353  0.563873
```

Here is a quick reference summary table of common functions. Each also takes an optional `level` parameter which applies only if the object has a *hierarchical index*.

Function	Description
`count`	Number of non-NA observations
`sum`	Sum of values
`mean`	Mean of values
`mad`	Mean absolute deviation
`median`	Arithmetic median of values
`min`	Minimum
`max`	Maximum
`mode`	Mode
`abs`	Absolute Value
`prod`	Product of values
`std`	Bessel-corrected sample standard deviation
`var`	Unbiased variance
`sem`	Standard error of the mean
`skew`	Sample skewness (3rd moment)
`kurt`	Sample kurtosis (4th moment)
`quantile`	Sample quantile (value at %)
`cumsum`	Cumulative sum
`cumprod`	Cumulative product
`cummax`	Cumulative maximum
`cummin`	Cumulative minimum

Note that by chance some NumPy methods, like `mean`, `std`, and `sum`, will exclude NAs on Series input by default:

```
In [88]: np.mean(df["one"])
Out[88]: 0.8110935116651192

In [89]: np.mean(df["one"].to_numpy())
Out[89]: nan
```

`Series.nunique()` will return the number of unique non-NA values in a Series:

```
In [90]: series = pd.Series(np.random.randn(500))

In [91]: series[20:500] = np.nan

In [92]: series[10:20] = 5

In [93]: series.nunique()
Out[93]: 11
```

Summarizing data: describe

There is a convenient *describe()* function which computes a variety of summary statistics about a Series or the columns of a DataFrame (excluding NAs of course):

```
In [94]: series = pd.Series(np.random.randn(1000))

In [95]: series[::2] = np.nan

In [96]: series.describe()
Out[96]:
count    500.000000
mean      -0.021292
std        1.015906
min       -2.683763
25%       -0.699070
50%       -0.069718
75%        0.714483
max        3.160915
dtype: float64

In [97]: frame = pd.DataFrame(np.random.randn(1000, 5), columns=["a", "b", "c", "d", "e
↪"])

In [98]: frame.iloc[::2] = np.nan

In [99]: frame.describe()
Out[99]:
                a           b           c           d           e
count  500.000000  500.000000  500.000000  500.000000  500.000000
mean     0.033387    0.030045   -0.043719   -0.051686    0.005979
std      1.017152    0.978743    1.025270    1.015988    1.006695
min     -3.000951   -2.637901   -3.303099   -3.159200   -3.188821
25%     -0.647623   -0.576449   -0.712369   -0.691338   -0.691115
50%      0.047578   -0.021499   -0.023888   -0.032652   -0.025363
75%      0.729907    0.775880    0.618896    0.670047    0.649748
max      2.740139    2.752332    3.004229    2.728702    3.240991
```

You can select specific percentiles to include in the output:

```
In [100]: series.describe(percentiles=[0.05, 0.25, 0.75, 0.95])
Out[100]:
count    500.000000
mean      -0.021292
std        1.015906
min       -2.683763
5%        -1.645423
25%       -0.699070
50%       -0.069718
75%        0.714483
95%        1.711409
max        3.160915
dtype: float64
```

By default, the median is always included.

For a non-numerical Series object, *describe()* will give a simple summary of the number of unique values and most frequently occurring values:

```
In [101]: s = pd.Series(["a", "a", "b", "b", "a", "a", np.nan, "c", "d", "a"])

In [102]: s.describe()
Out[102]:
count     9
unique    4
top       a
freq      5
dtype: object
```

Note that on a mixed-type DataFrame object, *describe()* will restrict the summary to include only numerical columns or, if none are, only categorical columns:

```
In [103]: frame = pd.DataFrame({"a": ["Yes", "Yes", "No", "No"], "b": range(4)})

In [104]: frame.describe()
Out[104]:
              b
count  4.000000
mean   1.500000
std    1.290994
min    0.000000
25%    0.750000
50%    1.500000
75%    2.250000
max    3.000000
```

This behavior can be controlled by providing a list of types as `include`/`exclude` arguments. The special value `all` can also be used:

```
In [105]: frame.describe(include=["object"])
Out[105]:
          a
count     4
unique    2
top     Yes
freq      2

In [106]: frame.describe(include=["number"])
Out[106]:
              b
count  4.000000
mean   1.500000
std    1.290994
min    0.000000
25%    0.750000
50%    1.500000
75%    2.250000
max    3.000000
```

(continues on next page)

(continued from previous page)

```
In [107]: frame.describe(include="all")
Out[107]:
          a         b
count     4  4.000000
unique    2       NaN
top     Yes       NaN
freq      2       NaN
mean    NaN  1.500000
std     NaN  1.290994
min     NaN  0.000000
25%     NaN  0.750000
50%     NaN  1.500000
75%     NaN  2.250000
max     NaN  3.000000
```

That feature relies on *select_dtypes*. Refer to there for details about accepted inputs.

Index of min/max values

The `idxmin()` and `idxmax()` functions on Series and DataFrame compute the index labels with the minimum and maximum corresponding values:

```
In [108]: s1 = pd.Series(np.random.randn(5))

In [109]: s1
Out[109]:
0    1.118076
1   -0.352051
2   -1.242883
3   -1.277155
4   -0.641184
dtype: float64

In [110]: s1.idxmin(), s1.idxmax()
Out[110]: (3, 0)

In [111]: df1 = pd.DataFrame(np.random.randn(5, 3), columns=["A", "B", "C"])

In [112]: df1
Out[112]:
          A         B         C
0 -0.327863 -0.946180 -0.137570
1 -0.186235 -0.257213 -0.486567
2 -0.507027 -0.871259 -0.111110
3  2.000339 -2.430505  0.089759
4 -0.321434 -0.033695  0.096271

In [113]: df1.idxmin(axis=0)
Out[113]:
A    2
```

(continues on next page)

(continued from previous page)

```
B    3
C    1
dtype: int64

In [114]: df1.idxmax(axis=1)
Out[114]:
0    C
1    A
2    C
3    A
4    C
dtype: object
```

When there are multiple rows (or columns) matching the minimum or maximum value, *idxmin()* and *idxmax()* return the first matching index:

```
In [115]: df3 = pd.DataFrame([2, 1, 1, 3, np.nan], columns=["A"], index=list("edcba"))

In [116]: df3
Out[116]:
     A
e  2.0
d  1.0
c  1.0
b  3.0
a  NaN

In [117]: df3["A"].idxmin()
Out[117]: 'd'
```

Note: `idxmin` and `idxmax` are called `argmin` and `argmax` in NumPy.

Value counts (histogramming) / mode

The *value_counts()* Series method and top-level function computes a histogram of a 1D array of values. It can also be used as a function on regular arrays:

```
In [118]: data = np.random.randint(0, 7, size=50)

In [119]: data
Out[119]:
array([6, 6, 2, 3, 5, 3, 2, 5, 4, 5, 4, 3, 4, 5, 0, 2, 0, 4, 2, 0, 3, 2,
       2, 5, 6, 5, 3, 4, 6, 4, 3, 5, 6, 4, 3, 6, 2, 6, 6, 2, 3, 4, 2, 1,
       6, 2, 6, 1, 5, 4])

In [120]: s = pd.Series(data)

In [121]: s.value_counts()
Out[121]:
6    10
```

(continues on next page)

(continued from previous page)

```
2    10
4     9
3     8
5     8
0     3
1     2
dtype: int64

In [122]: pd.value_counts(data)
Out[122]:
6    10
2    10
4     9
3     8
5     8
0     3
1     2
dtype: int64
```

New in version 1.1.0.

The `value_counts()` method can be used to count combinations across multiple columns. By default all columns are used but a subset can be selected using the `subset` argument.

```
In [123]: data = {"a": [1, 2, 3, 4], "b": ["x", "x", "y", "y"]}

In [124]: frame = pd.DataFrame(data)

In [125]: frame.value_counts()
Out[125]:
a  b
1  x    1
2  x    1
3  y    1
4  y    1
dtype: int64
```

Similarly, you can get the most frequently occurring value(s), i.e. the mode, of the values in a Series or DataFrame:

```
In [126]: s5 = pd.Series([1, 1, 3, 3, 3, 5, 5, 7, 7, 7])

In [127]: s5.mode()
Out[127]:
0    3
1    7
dtype: int64

In [128]: df5 = pd.DataFrame(
   .....:     {
   .....:         "A": np.random.randint(0, 7, size=50),
   .....:         "B": np.random.randint(-10, 15, size=50),
   .....:     }
   .....: )
```

(continues on next page)

(continued from previous page)

```
   .....:

In [129]: df5.mode()
Out[129]: 
     A   B
0  1.0  -9
1  NaN  10
2  NaN  13
```

Discretization and quantiling

Continuous values can be discretized using the *cut()* (bins based on values) and *qcut()* (bins based on sample quantiles) functions:

```
In [130]: arr = np.random.randn(20)

In [131]: factor = pd.cut(arr, 4)

In [132]: factor
Out[132]: 
[(-0.251, 0.464], (-0.968, -0.251], (0.464, 1.179], (-0.251, 0.464], (-0.968, -0.251], ..
↪., (-0.251, 0.464], (-0.968, -0.251], (-0.968, -0.251], (-0.968, -0.251], (-0.968, -0.
↪251]]
Length: 20
Categories (4, interval[float64, right]): [(-0.968, -0.251] < (-0.251, 0.464] < (0.464,␣
↪1.179] <
                                           (1.179, 1.893]]

In [133]: factor = pd.cut(arr, [-5, -1, 0, 1, 5])

In [134]: factor
Out[134]: 
[(0, 1], (-1, 0], (0, 1], (0, 1], (-1, 0], ..., (-1, 0], (-1, 0], (-1, 0], (-1, 0], (-1,␣
↪0]]
Length: 20
Categories (4, interval[int64, right]): [(-5, -1] < (-1, 0] < (0, 1] < (1, 5]]
```

qcut() computes sample quantiles. For example, we could slice up some normally distributed data into equal-size quartiles like so:

```
In [135]: arr = np.random.randn(30)

In [136]: factor = pd.qcut(arr, [0, 0.25, 0.5, 0.75, 1])

In [137]: factor
Out[137]: 
[(0.569, 1.184], (-2.278, -0.301], (-2.278, -0.301], (0.569, 1.184], (0.569, 1.184], ...,
↪ (-0.301, 0.569], (1.184, 2.346], (1.184, 2.346], (-0.301, 0.569], (-2.278, -0.301]]
Length: 30
Categories (4, interval[float64, right]): [(-2.278, -0.301] < (-0.301, 0.569] < (0.569,␣
↪1.184] <
```

(continues on next page)

(continued from previous page)

```
                                            (1.184, 2.346]]

In [138]: pd.value_counts(factor)
Out[138]: 
(-2.278, -0.301]    8
(1.184, 2.346]      8
(-0.301, 0.569]     7
(0.569, 1.184]      7
dtype: int64
```

We can also pass infinite values to define the bins:

```
In [139]: arr = np.random.randn(20)

In [140]: factor = pd.cut(arr, [-np.inf, 0, np.inf])

In [141]: factor
Out[141]: 
[(-inf, 0.0], (0.0, inf], (0.0, inf], (-inf, 0.0], (-inf, 0.0], ..., (-inf, 0.0], (-inf,␣
↪0.0], (-inf, 0.0], (0.0, inf], (0.0, inf]]
Length: 20
Categories (2, interval[float64, right]): [(-inf, 0.0] < (0.0, inf]]
```

2.3.6 Function application

To apply your own or another library's functions to pandas objects, you should be aware of the three methods below. The appropriate method to use depends on whether your function expects to operate on an entire `DataFrame` or `Series`, row- or column-wise, or elementwise.

1. *Tablewise Function Application*: `pipe()`
2. *Row or Column-wise Function Application*: `apply()`
3. *Aggregation API*: `agg()` and `transform()`
4. *Applying Elementwise Functions*: `applymap()`

Tablewise function application

`DataFrames` and `Series` can be passed into functions. However, if the function needs to be called in a chain, consider using the `pipe()` method.

First some setup:

```
In [142]: def extract_city_name(df):
   .....:     """
   .....:     Chicago, IL -> Chicago for city_name column
   .....:     """
   .....:     df["city_name"] = df["city_and_code"].str.split(",").str.get(0)
   .....:     return df
   .....: 

In [143]: def add_country_name(df, country_name=None):
```

(continues on next page)

(continued from previous page)

```
   .....:     """
   .....:     Chicago -> Chicago-US for city_name column
   .....:     """
   .....:     col = "city_name"
   .....:     df["city_and_country"] = df[col] + country_name
   .....:     return df
   .....: 

In [144]: df_p = pd.DataFrame({"city_and_code": ["Chicago, IL"]})
```

`extract_city_name` and `add_country_name` are functions taking and returning `DataFrames`.

Now compare the following:

```
In [145]: add_country_name(extract_city_name(df_p), country_name="US")
Out[145]: 
  city_and_code city_name city_and_country
0   Chicago, IL   Chicago        ChicagoUS
```

Is equivalent to:

```
In [146]: df_p.pipe(extract_city_name).pipe(add_country_name, country_name="US")
Out[146]: 
  city_and_code city_name city_and_country
0   Chicago, IL   Chicago        ChicagoUS
```

pandas encourages the second style, which is known as method chaining. `pipe` makes it easy to use your own or another library's functions in method chains, alongside pandas' methods.

In the example above, the functions `extract_city_name` and `add_country_name` each expected a `DataFrame` as the first positional argument. What if the function you wish to apply takes its data as, say, the second argument? In this case, provide `pipe` with a tuple of `(callable, data_keyword)`. `.pipe` will route the `DataFrame` to the argument specified in the tuple.

For example, we can fit a regression using statsmodels. Their API expects a formula first and a `DataFrame` as the second argument, `data`. We pass in the function, keyword pair `(sm.ols, 'data')` to `pipe`:

```
In [147]: import statsmodels.formula.api as sm

In [148]: bb = pd.read_csv("data/baseball.csv", index_col="id")

In [149]: (
   .....:     bb.query("h > 0")
   .....:     .assign(ln_h=lambda df: np.log(df.h))
   .....:     .pipe((sm.ols, "data"), "hr ~ ln_h + year + g + C(lg)")
   .....:     .fit()
   .....:     .summary()
   .....: )
   .....: 
Out[149]: 
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
```

(continues on next page)

(continued from previous page)

```
Dep. Variable:                     hr   R-squared:                       0.685
Model:                            OLS   Adj. R-squared:                  0.665
Method:                 Least Squares   F-statistic:                     34.28
Date:                Sat, 22 Jan 2022   Prob (F-statistic):           3.48e-15
Time:                        10:50:02   Log-Likelihood:                -205.92
No. Observations:                  68   AIC:                             421.8
Df Residuals:                      63   BIC:                             432.9
Df Model:                           4
Covariance Type:            nonrobust
===============================================================================
                  coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------
Intercept   -8484.7720   4664.146     -1.819      0.074   -1.78e+04     835.780
C(lg)[T.NL]    -2.2736      1.325     -1.716      0.091      -4.922       0.375
ln_h           -1.3542      0.875     -1.547      0.127      -3.103       0.395
year            4.2277      2.324      1.819      0.074      -0.417       8.872
g               0.1841      0.029      6.258      0.000       0.125       0.243
==============================================================================
Omnibus:                       10.875   Durbin-Watson:                   1.999
Prob(Omnibus):                  0.004   Jarque-Bera (JB):               17.298
Skew:                           0.537   Prob(JB):                     0.000175
Kurtosis:                       5.225   Cond. No.                     1.49e+07
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
[2] The condition number is large, 1.49e+07. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

The pipe method is inspired by unix pipes and more recently dplyr and magrittr, which have introduced the popular (%>%) (read pipe) operator for R. The implementation of pipe here is quite clean and feels right at home in Python. We encourage you to view the source code of *pipe()*.

Row or column-wise function application

Arbitrary functions can be applied along the axes of a DataFrame using the *apply()* method, which, like the descriptive statistics methods, takes an optional axis argument:

```
In [150]: df.apply(np.mean)
Out[150]:
one      0.811094
two      1.360588
three    0.187958
dtype: float64

In [151]: df.apply(np.mean, axis=1)
Out[151]:
a    1.583749
b    0.734929
```

(continues on next page)

(continued from previous page)

```
c    1.133683
d   -0.166914
dtype: float64

In [152]: df.apply(lambda x: x.max() - x.min())
Out[152]:
one      1.051928
two      1.632779
three    1.840607
dtype: float64

In [153]: df.apply(np.cumsum)
Out[153]:
        one       two     three
a  1.394981  1.772517       NaN
b  1.738035  3.684640 -0.050390
c  2.433281  5.163008  1.177045
d       NaN  5.442353  0.563873

In [154]: df.apply(np.exp)
Out[154]:
        one       two     three
a  4.034899  5.885648       NaN
b  1.409244  6.767440  0.950858
c  2.004201  4.385785  3.412466
d       NaN  1.322262  0.541630
```

The *apply()* method will also dispatch on a string method name.

```
In [155]: df.apply("mean")
Out[155]:
one      0.811094
two      1.360588
three    0.187958
dtype: float64

In [156]: df.apply("mean", axis=1)
Out[156]:
a    1.583749
b    0.734929
c    1.133683
d   -0.166914
dtype: float64
```

The return type of the function passed to *apply()* affects the type of the final output from `DataFrame.apply` for the default behaviour:

- If the applied function returns a `Series`, the final output is a `DataFrame`. The columns match the index of the `Series` returned by the applied function.
- If the applied function returns any other type, the final output is a `Series`.

This default behaviour can be overridden using the `result_type`, which accepts three options: `reduce`, `broadcast`, and `expand`. These will determine how list-likes return values expand (or not) to a `DataFrame`.

apply() combined with some cleverness can be used to answer many questions about a data set. For example, suppose we wanted to extract the date where the maximum value for each column occurred:

```
In [157]: tsdf = pd.DataFrame(
   .....:     np.random.randn(1000, 3),
   .....:     columns=["A", "B", "C"],
   .....:     index=pd.date_range("1/1/2000", periods=1000),
   .....: )
   .....:

In [158]: tsdf.apply(lambda x: x.idxmax())
Out[158]:
A   2000-08-06
B   2001-01-18
C   2001-07-18
dtype: datetime64[ns]
```

You may also pass additional arguments and keyword arguments to the apply() method. For instance, consider the following function you would like to apply:

```
def subtract_and_divide(x, sub, divide=1):
    return (x - sub) / divide
```

You may then apply this function as follows:

```
df.apply(subtract_and_divide, args=(5,), divide=3)
```

Another useful feature is the ability to pass Series methods to carry out some Series operation on each column or row:

```
In [159]: tsdf
Out[159]:
                   A         B         C
2000-01-01 -0.158131 -0.232466  0.321604
2000-01-02 -1.810340 -3.105758  0.433834
2000-01-03 -1.209847 -1.156793 -0.136794
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08 -0.653602  0.178875  1.008298
2000-01-09  1.007996  0.462824  0.254472
2000-01-10  0.307473  0.600337  1.643950

In [160]: tsdf.apply(pd.Series.interpolate)
Out[160]:
                   A         B         C
2000-01-01 -0.158131 -0.232466  0.321604
2000-01-02 -1.810340 -3.105758  0.433834
2000-01-03 -1.209847 -1.156793 -0.136794
2000-01-04 -1.098598 -0.889659  0.092225
2000-01-05 -0.987349 -0.622526  0.321243
2000-01-06 -0.876100 -0.355392  0.550262
2000-01-07 -0.764851 -0.088259  0.779280
2000-01-08 -0.653602  0.178875  1.008298
```

(continues on next page)

(continued from previous page)

```
2000-01-09  1.007996  0.462824  0.254472
2000-01-10  0.307473  0.600337  1.643950
```

Finally, *apply()* takes an argument `raw` which is False by default, which converts each row or column into a Series before applying the function. When set to True, the passed function will instead receive an ndarray object, which has positive performance implications if you do not need the indexing functionality.

Aggregation API

The aggregation API allows one to express possibly multiple aggregation operations in a single concise way. This API is similar across pandas objects, see *groupby API*, the *window API*, and the *resample API*. The entry point for aggregation is *DataFrame.aggregate()*, or the alias *DataFrame.agg()*.

We will use a similar starting frame from above:

```
In [161]: tsdf = pd.DataFrame(
   .....:     np.random.randn(10, 3),
   .....:     columns=["A", "B", "C"],
   .....:     index=pd.date_range("1/1/2000", periods=10),
   .....: )
   .....:

In [162]: tsdf.iloc[3:7] = np.nan

In [163]: tsdf
Out[163]:
                   A         B         C
2000-01-01  1.257606  1.004194  0.167574
2000-01-02 -0.749892  0.288112 -0.757304
2000-01-03 -0.207550 -0.298599  0.116018
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.814347 -0.257623  0.869226
2000-01-09 -0.250663 -1.206601  0.896839
2000-01-10  2.169758 -1.333363  0.283157
```

Using a single function is equivalent to *apply()*. You can also pass named methods as strings. These will return a `Series` of the aggregated output:

```
In [164]: tsdf.agg(np.sum)
Out[164]:
A    3.033606
B   -1.803879
C    1.575510
dtype: float64

In [165]: tsdf.agg("sum")
Out[165]:
A    3.033606
B   -1.803879
```

(continues on next page)

(continued from previous page)

```
C    1.575510
dtype: float64

# these are equivalent to a ``.sum()`` because we are aggregating
# on a single function
In [166]: tsdf.sum()
Out[166]:
A    3.033606
B   -1.803879
C    1.575510
dtype: float64
```

Single aggregations on a `Series` this will return a scalar value:

```
In [167]: tsdf["A"].agg("sum")
Out[167]: 3.033606102414146
```

Aggregating with multiple functions

You can pass multiple aggregation arguments as a list. The results of each of the passed functions will be a row in the resulting `DataFrame`. These are naturally named from the aggregation function.

```
In [168]: tsdf.agg(["sum"])
Out[168]:
            A         B        C
sum  3.033606 -1.803879  1.57551
```

Multiple functions yield multiple rows:

```
In [169]: tsdf.agg(["sum", "mean"])
Out[169]:
             A         B         C
sum   3.033606 -1.803879  1.575510
mean  0.505601 -0.300647  0.262585
```

On a `Series`, multiple functions return a `Series`, indexed by the function names:

```
In [170]: tsdf["A"].agg(["sum", "mean"])
Out[170]:
sum     3.033606
mean    0.505601
Name: A, dtype: float64
```

Passing a `lambda` function will yield a `<lambda>` named row:

```
In [171]: tsdf["A"].agg(["sum", lambda x: x.mean()])
Out[171]:
sum         3.033606
<lambda>    0.505601
Name: A, dtype: float64
```

Passing a named function will yield that name for the row:

```
In [172]: def mymean(x):
   .....:     return x.mean()
   .....:

In [173]: tsdf["A"].agg(["sum", mymean])
Out[173]:
sum       3.033606
mymean    0.505601
Name: A, dtype: float64
```

Aggregating with a dict

Passing a dictionary of column names to a scalar or a list of scalars, to `DataFrame.agg` allows you to customize which functions are applied to which columns. Note that the results are not in any particular order, you can use an `OrderedDict` instead to guarantee ordering.

```
In [174]: tsdf.agg({"A": "mean", "B": "sum"})
Out[174]:
A    0.505601
B   -1.803879
dtype: float64
```

Passing a list-like will generate a `DataFrame` output. You will get a matrix-like output of all of the aggregators. The output will consist of all unique functions. Those that are not noted for a particular column will be `NaN`:

```
In [175]: tsdf.agg({"A": ["mean", "min"], "B": "sum"})
Out[175]:
             A         B
mean  0.505601       NaN
min  -0.749892       NaN
sum        NaN -1.803879
```

Mixed dtypes

Deprecated since version 1.4.0: Attempting to determine which columns cannot be aggregated and silently dropping them from the results is deprecated and will be removed in a future version. If any porition of the columns or operations provided fail, the call to `.agg` will raise.

When presented with mixed dtypes that cannot aggregate, `.agg` will only take the valid aggregations. This is similar to how `.groupby.agg` works.

```
In [176]: mdf = pd.DataFrame(
   .....:     {
   .....:         "A": [1, 2, 3],
   .....:         "B": [1.0, 2.0, 3.0],
   .....:         "C": ["foo", "bar", "baz"],
   .....:         "D": pd.date_range("20130101", periods=3),
   .....:     }
   .....: )
   .....:
```

(continues on next page)

(continued from previous page)

```
In [177]: mdf.dtypes
Out[177]:
A              int64
B            float64
C             object
D     datetime64[ns]
dtype: object
```

```
In [178]: mdf.agg(["min", "sum"])
Out[178]:
     A    B          C          D
min  1  1.0        bar 2013-01-01
sum  6  6.0  foobarbaz        NaT
```

Custom describe

With `.agg()` it is possible to easily create a custom describe function, similar to the built in *describe function*.

```
In [179]: from functools import partial

In [180]: q_25 = partial(pd.Series.quantile, q=0.25)

In [181]: q_25.__name__ = "25%"

In [182]: q_75 = partial(pd.Series.quantile, q=0.75)

In [183]: q_75.__name__ = "75%"

In [184]: tsdf.agg(["count", "mean", "std", "min", q_25, "median", q_75, "max"])
Out[184]:
               A         B         C
count   6.000000  6.000000  6.000000
mean    0.505601 -0.300647  0.262585
std     1.103362  0.887508  0.606860
min    -0.749892 -1.333363 -0.757304
25%    -0.239885 -0.979600  0.128907
median  0.303398 -0.278111  0.225365
75%     1.146791  0.151678  0.722709
max     2.169758  1.004194  0.896839
```

Transform API

The `transform()` method returns an object that is indexed the same (same size) as the original. This API allows you to provide *multiple* operations at the same time rather than one-by-one. Its API is quite similar to the `.agg` API.

We create a frame similar to the one used in the above sections.

```
In [185]: tsdf = pd.DataFrame(
   .....:     np.random.randn(10, 3),
   .....:     columns=["A", "B", "C"],
```

(continues on next page)

(continued from previous page)

```
   .....:     index=pd.date_range("1/1/2000", periods=10),
   .....: )
   .....:

In [186]: tsdf.iloc[3:7] = np.nan

In [187]: tsdf
Out[187]:
                   A         B         C
2000-01-01 -0.428759 -0.864890 -0.675341
2000-01-02 -0.168731  1.338144 -1.279321
2000-01-03 -1.621034  0.438107  0.903794
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.254374 -1.240447 -0.201052
2000-01-09 -0.157795  0.791197 -1.144209
2000-01-10 -0.030876  0.371900  0.061932
```

Transform the entire frame. `.transform()` allows input functions as: a NumPy function, a string function name or a user defined function.

```
In [188]: tsdf.transform(np.abs)
Out[188]:
                   A         B         C
2000-01-01  0.428759  0.864890  0.675341
2000-01-02  0.168731  1.338144  1.279321
2000-01-03  1.621034  0.438107  0.903794
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.254374  1.240447  0.201052
2000-01-09  0.157795  0.791197  1.144209
2000-01-10  0.030876  0.371900  0.061932

In [189]: tsdf.transform("abs")
Out[189]:
                   A         B         C
2000-01-01  0.428759  0.864890  0.675341
2000-01-02  0.168731  1.338144  1.279321
2000-01-03  1.621034  0.438107  0.903794
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.254374  1.240447  0.201052
2000-01-09  0.157795  0.791197  1.144209
2000-01-10  0.030876  0.371900  0.061932

In [190]: tsdf.transform(lambda x: x.abs())
```

(continues on next page)

(continued from previous page)

```
Out[190]:
                   A         B         C
2000-01-01  0.428759  0.864890  0.675341
2000-01-02  0.168731  1.338144  1.279321
2000-01-03  1.621034  0.438107  0.903794
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.254374  1.240447  0.201052
2000-01-09  0.157795  0.791197  1.144209
2000-01-10  0.030876  0.371900  0.061932
```

Here `transform()` received a single function; this is equivalent to a ufunc application.

```
In [191]: np.abs(tsdf)
Out[191]:
                   A         B         C
2000-01-01  0.428759  0.864890  0.675341
2000-01-02  0.168731  1.338144  1.279321
2000-01-03  1.621034  0.438107  0.903794
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.254374  1.240447  0.201052
2000-01-09  0.157795  0.791197  1.144209
2000-01-10  0.030876  0.371900  0.061932
```

Passing a single function to `.transform()` with a `Series` will yield a single `Series` in return.

```
In [192]: tsdf["A"].transform(np.abs)
Out[192]:
2000-01-01    0.428759
2000-01-02    0.168731
2000-01-03    1.621034
2000-01-04         NaN
2000-01-05         NaN
2000-01-06         NaN
2000-01-07         NaN
2000-01-08    0.254374
2000-01-09    0.157795
2000-01-10    0.030876
Freq: D, Name: A, dtype: float64
```

Transform with multiple functions

Passing multiple functions will yield a column MultiIndexed DataFrame. The first level will be the original frame column names; the second level will be the names of the transforming functions.

```
In [193]: tsdf.transform([np.abs, lambda x: x + 1])
Out[193]:
                   A                   B                   C
            absolute  <lambda>  absolute  <lambda>  absolute  <lambda>
2000-01-01  0.428759  0.571241  0.864890  0.135110  0.675341  0.324659
2000-01-02  0.168731  0.831269  1.338144  2.338144  1.279321 -0.279321
2000-01-03  1.621034 -0.621034  0.438107  1.438107  0.903794  1.903794
2000-01-04       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN       NaN       NaN       NaN
2000-01-08  0.254374  1.254374  1.240447 -0.240447  0.201052  0.798948
2000-01-09  0.157795  0.842205  0.791197  1.791197  1.144209 -0.144209
2000-01-10  0.030876  0.969124  0.371900  1.371900  0.061932  1.061932
```

Passing multiple functions to a Series will yield a DataFrame. The resulting column names will be the transforming functions.

```
In [194]: tsdf["A"].transform([np.abs, lambda x: x + 1])
Out[194]:
            absolute  <lambda>
2000-01-01  0.428759  0.571241
2000-01-02  0.168731  0.831269
2000-01-03  1.621034 -0.621034
2000-01-04       NaN       NaN
2000-01-05       NaN       NaN
2000-01-06       NaN       NaN
2000-01-07       NaN       NaN
2000-01-08  0.254374  1.254374
2000-01-09  0.157795  0.842205
2000-01-10  0.030876  0.969124
```

Transforming with a dict

Passing a dict of functions will allow selective transforming per column.

```
In [195]: tsdf.transform({"A": np.abs, "B": lambda x: x + 1})
Out[195]:
                   A         B
2000-01-01  0.428759  0.135110
2000-01-02  0.168731  2.338144
2000-01-03  1.621034  1.438107
2000-01-04       NaN       NaN
2000-01-05       NaN       NaN
2000-01-06       NaN       NaN
2000-01-07       NaN       NaN
2000-01-08  0.254374 -0.240447
```

(continues on next page)

(continued from previous page)

```
2000-01-09  0.157795  1.791197
2000-01-10  0.030876  1.371900
```

Passing a dict of lists will generate a MultiIndexed DataFrame with these selective transforms.

```
In [196]: tsdf.transform({"A": np.abs, "B": [lambda x: x + 1, "sqrt"]})
Out[196]:
                   A         B
            absolute  <lambda>      sqrt
2000-01-01  0.428759  0.135110       NaN
2000-01-02  0.168731  2.338144  1.156782
2000-01-03  1.621034  1.438107  0.661897
2000-01-04       NaN       NaN       NaN
2000-01-05       NaN       NaN       NaN
2000-01-06       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN
2000-01-08  0.254374 -0.240447       NaN
2000-01-09  0.157795  1.791197  0.889493
2000-01-10  0.030876  1.371900  0.609836
```

Applying elementwise functions

Since not all functions can be vectorized (accept NumPy arrays and return another array or value), the methods *applymap()* on DataFrame and analogously *map()* on Series accept any Python function taking a single value and returning a single value. For example:

```
In [197]: df4
Out[197]:
        one       two     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [198]: def f(x):
   .....:     return len(str(x))
   .....:

In [199]: df4["one"].map(f)
Out[199]:
a    18
b    19
c    18
d     3
Name: one, dtype: int64

In [200]: df4.applymap(f)
Out[200]:
   one  two  three
a   18   17      3
b   19   18     20
```

(continues on next page)

(continued from previous page)

```
c    18    18      16
d     3    19      19
```

Series.map() has an additional feature; it can be used to easily "link" or "map" values defined by a secondary series. This is closely related to *merging/joining functionality*:

```
In [201]: s = pd.Series(
   .....:     ["six", "seven", "six", "seven", "six"], index=["a", "b", "c", "d", "e"]
   .....: )
   .....:

In [202]: t = pd.Series({"six": 6.0, "seven": 7.0})

In [203]: s
Out[203]:
a      six
b    seven
c      six
d    seven
e      six
dtype: object

In [204]: s.map(t)
Out[204]:
a    6.0
b    7.0
c    6.0
d    7.0
e    6.0
dtype: float64
```

2.3.7 Reindexing and altering labels

reindex() is the fundamental data alignment method in pandas. It is used to implement nearly all other features relying on label-alignment functionality. To *reindex* means to conform the data to match a given set of labels along a particular axis. This accomplishes several things:

- Reorders the existing data to match a new set of labels
- Inserts missing value (NA) markers in label locations where no data for that label existed
- If specified, **fill** data for missing labels using logic (highly relevant to working with time series data)

Here is a simple example:

```
In [205]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])

In [206]: s
Out[206]:
a    1.695148
b    1.328614
c    1.234686
d   -0.385845
```

(continues on next page)

(continued from previous page)

```
e   -1.326508
dtype: float64

In [207]: s.reindex(["e", "b", "f", "d"])
Out[207]:
e   -1.326508
b    1.328614
f         NaN
d   -0.385845
dtype: float64
```

Here, the `f` label was not contained in the Series and hence appears as `NaN` in the result.

With a DataFrame, you can simultaneously reindex the index and columns:

```
In [208]: df
Out[208]:
        one       two     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [209]: df.reindex(index=["c", "f", "b"], columns=["three", "two", "one"])
Out[209]:
      three       two       one
c  1.227435  1.478369  0.695246
f       NaN       NaN       NaN
b -0.050390  1.912123  0.343054
```

You may also use `reindex` with an `axis` keyword:

```
In [210]: df.reindex(["c", "f", "b"], axis="index")
Out[210]:
        one       two     three
c  0.695246  1.478369  1.227435
f       NaN       NaN       NaN
b  0.343054  1.912123 -0.050390
```

Note that the `Index` objects containing the actual axis labels can be **shared** between objects. So if we have a Series and a DataFrame, the following can be done:

```
In [211]: rs = s.reindex(df.index)

In [212]: rs
Out[212]:
a    1.695148
b    1.328614
c    1.234686
d   -0.385845
dtype: float64

In [213]: rs.index is df.index
```

(continues on next page)

(continued from previous page)

```
Out[213]: True
```

This means that the reindexed Series's index is the same Python object as the DataFrame's index.

DataFrame.reindex() also supports an "axis-style" calling convention, where you specify a single `labels` argument and the `axis` it applies to.

```
In [214]: df.reindex(["c", "f", "b"], axis="index")
Out[214]:
        one       two     three
c  0.695246  1.478369  1.227435
f       NaN       NaN       NaN
b  0.343054  1.912123 -0.050390

In [215]: df.reindex(["three", "two", "one"], axis="columns")
Out[215]:
      three       two       one
a       NaN  1.772517  1.394981
b -0.050390  1.912123  0.343054
c  1.227435  1.478369  0.695246
d -0.613172  0.279344       NaN
```

See also:

MultiIndex / Advanced Indexing is an even more concise way of doing reindexing.

Note: When writing performance-sensitive code, there is a good reason to spend some time becoming a reindexing ninja: **many operations are faster on pre-aligned data**. Adding two unaligned DataFrames internally triggers a reindexing step. For exploratory analysis you will hardly notice the difference (because `reindex` has been heavily optimized), but when CPU cycles matter sprinkling a few explicit `reindex` calls here and there can have an impact.

Reindexing to align with another object

You may wish to take an object and reindex its axes to be labeled the same as another object. While the syntax for this is straightforward albeit verbose, it is a common enough operation that the *reindex_like()* method is available to make this simpler:

```
In [216]: df2
Out[216]:
        one       two
a  1.394981  1.772517
b  0.343054  1.912123
c  0.695246  1.478369

In [217]: df3
Out[217]:
        one       two
a  0.583888  0.051514
b -0.468040  0.191120
c -0.115848 -0.242634
```

(continues on next page)

(continued from previous page)

```
In [218]: df.reindex_like(df2)
Out[218]:
        one       two
a  1.394981  1.772517
b  0.343054  1.912123
c  0.695246  1.478369
```

Aligning objects with each other with `align`

The *align()* method is the fastest way to simultaneously align two objects. It supports a `join` argument (related to *joining and merging*):

- `join='outer'`: take the union of the indexes (default)
- `join='left'`: use the calling object's index
- `join='right'`: use the passed object's index
- `join='inner'`: intersect the indexes

It returns a tuple with both of the reindexed Series:

```
In [219]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])

In [220]: s1 = s[:4]

In [221]: s2 = s[1:]

In [222]: s1.align(s2)
Out[222]:
(a   -0.186646
 b   -1.692424
 c   -0.303893
 d   -1.425662
 e         NaN
 dtype: float64,
 a         NaN
 b   -1.692424
 c   -0.303893
 d   -1.425662
 e    1.114285
 dtype: float64)

In [223]: s1.align(s2, join="inner")
Out[223]:
(b   -1.692424
 c   -0.303893
 d   -1.425662
 dtype: float64,
 b   -1.692424
 c   -0.303893
 d   -1.425662
 dtype: float64)
```

(continues on next page)

(continued from previous page)

```
In [224]: s1.align(s2, join="left")
Out[224]:
(a   -0.186646
 b   -1.692424
 c   -0.303893
 d   -1.425662
 dtype: float64,
 a         NaN
 b   -1.692424
 c   -0.303893
 d   -1.425662
 dtype: float64)
```

For DataFrames, the join method will be applied to both the index and the columns by default:

```
In [225]: df.align(df2, join="inner")
Out[225]:
(        one       two
 a  1.394981  1.772517
 b  0.343054  1.912123
 c  0.695246  1.478369,
         one       two
 a  1.394981  1.772517
 b  0.343054  1.912123
 c  0.695246  1.478369)
```

You can also pass an `axis` option to only align on the specified axis:

```
In [226]: df.align(df2, join="inner", axis=0)
Out[226]:
(        one       two     three
 a  1.394981  1.772517       NaN
 b  0.343054  1.912123 -0.050390
 c  0.695246  1.478369  1.227435,
         one       two
 a  1.394981  1.772517
 b  0.343054  1.912123
 c  0.695246  1.478369)
```

If you pass a Series to *DataFrame.align()*, you can choose to align both objects either on the DataFrame's index or columns using the `axis` argument:

```
In [227]: df.align(df2.iloc[0], axis=1)
Out[227]:
(        one     three       two
 a  1.394981       NaN  1.772517
 b  0.343054 -0.050390  1.912123
 c  0.695246  1.227435  1.478369
 d       NaN -0.613172  0.279344,
 one       1.394981
 three          NaN
```

(continues on next page)

(continued from previous page)

```
two      1.772517
Name: a, dtype: float64)
```

Filling while reindexing

reindex() takes an optional parameter `method` which is a filling method chosen from the following table:

Method	Action
pad / ffill	Fill values forward
bfill / backfill	Fill values backward
nearest	Fill from the nearest index value

We illustrate these fill methods on a simple Series:

```
In [228]: rng = pd.date_range("1/3/2000", periods=8)

In [229]: ts = pd.Series(np.random.randn(8), index=rng)

In [230]: ts2 = ts[[0, 3, 6]]

In [231]: ts
Out[231]:
2000-01-03    0.183051
2000-01-04    0.400528
2000-01-05   -0.015083
2000-01-06    2.395489
2000-01-07    1.414806
2000-01-08    0.118428
2000-01-09    0.733639
2000-01-10   -0.936077
Freq: D, dtype: float64

In [232]: ts2
Out[232]:
2000-01-03    0.183051
2000-01-06    2.395489
2000-01-09    0.733639
Freq: 3D, dtype: float64

In [233]: ts2.reindex(ts.index)
Out[233]:
2000-01-03    0.183051
2000-01-04         NaN
2000-01-05         NaN
2000-01-06    2.395489
2000-01-07         NaN
2000-01-08         NaN
2000-01-09    0.733639
2000-01-10         NaN
Freq: D, dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [234]: ts2.reindex(ts.index, method="ffill")
Out[234]:
2000-01-03    0.183051
2000-01-04    0.183051
2000-01-05    0.183051
2000-01-06    2.395489
2000-01-07    2.395489
2000-01-08    2.395489
2000-01-09    0.733639
2000-01-10    0.733639
Freq: D, dtype: float64

In [235]: ts2.reindex(ts.index, method="bfill")
Out[235]:
2000-01-03    0.183051
2000-01-04    2.395489
2000-01-05    2.395489
2000-01-06    2.395489
2000-01-07    0.733639
2000-01-08    0.733639
2000-01-09    0.733639
2000-01-10         NaN
Freq: D, dtype: float64

In [236]: ts2.reindex(ts.index, method="nearest")
Out[236]:
2000-01-03    0.183051
2000-01-04    0.183051
2000-01-05    2.395489
2000-01-06    2.395489
2000-01-07    2.395489
2000-01-08    0.733639
2000-01-09    0.733639
2000-01-10    0.733639
Freq: D, dtype: float64
```

These methods require that the indexes are **ordered** increasing or decreasing.

Note that the same result could have been achieved using *fillna* (except for `method='nearest'`) or *interpolate*:

```
In [237]: ts2.reindex(ts.index).fillna(method="ffill")
Out[237]:
2000-01-03    0.183051
2000-01-04    0.183051
2000-01-05    0.183051
2000-01-06    2.395489
2000-01-07    2.395489
2000-01-08    2.395489
2000-01-09    0.733639
2000-01-10    0.733639
Freq: D, dtype: float64
```

reindex() will raise a ValueError if the index is not monotonically increasing or decreasing. *fillna()* and *interpolate()* will not perform any checks on the order of the index.

Limits on filling while reindexing

The `limit` and `tolerance` arguments provide additional control over filling while reindexing. Limit specifies the maximum count of consecutive matches:

```
In [238]: ts2.reindex(ts.index, method="ffill", limit=1)
Out[238]:
2000-01-03    0.183051
2000-01-04    0.183051
2000-01-05         NaN
2000-01-06    2.395489
2000-01-07    2.395489
2000-01-08         NaN
2000-01-09    0.733639
2000-01-10    0.733639
Freq: D, dtype: float64
```

In contrast, tolerance specifies the maximum distance between the index and indexer values:

```
In [239]: ts2.reindex(ts.index, method="ffill", tolerance="1 day")
Out[239]:
2000-01-03    0.183051
2000-01-04    0.183051
2000-01-05         NaN
2000-01-06    2.395489
2000-01-07    2.395489
2000-01-08         NaN
2000-01-09    0.733639
2000-01-10    0.733639
Freq: D, dtype: float64
```

Notice that when used on a `DatetimeIndex`, `TimedeltaIndex` or `PeriodIndex`, `tolerance` will coerced into a `Timedelta` if possible. This allows you to specify tolerance with appropriate strings.

Dropping labels from an axis

A method closely related to `reindex` is the *drop()* function. It removes a set of labels from an axis:

```
In [240]: df
Out[240]:
        one       two     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [241]: df.drop(["a", "d"], axis=0)
Out[241]:
        one       two     three
```

(continues on next page)

(continued from previous page)

```
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435

In [242]: df.drop(["one"], axis=1)
Out[242]:
        two     three
a  1.772517       NaN
b  1.912123 -0.050390
c  1.478369  1.227435
d  0.279344 -0.613172
```

Note that the following also works, but is a bit less obvious / clean:

```
In [243]: df.reindex(df.index.difference(["a", "d"]))
Out[243]:
        one       two     three
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
```

Renaming / mapping labels

The *rename()* method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

```
In [244]: s
Out[244]:
a   -0.186646
b   -1.692424
c   -0.303893
d   -1.425662
e    1.114285
dtype: float64

In [245]: s.rename(str.upper)
Out[245]:
A   -0.186646
B   -1.692424
C   -0.303893
D   -1.425662
E    1.114285
dtype: float64
```

If you pass a function, it must return a value when called with any of the labels (and must produce a set of unique values). A dict or Series can also be used:

```
In [246]: df.rename(
   .....:     columns={"one": "foo", "two": "bar"},
   .....:     index={"a": "apple", "b": "banana", "d": "durian"},
   .....: )
   .....:
Out[246]:
             foo       bar     three
```

(continues on next page)

(continued from previous page)

```
apple   1.394981  1.772517       NaN
banana  0.343054  1.912123 -0.050390
c       0.695246  1.478369  1.227435
durian       NaN  0.279344 -0.613172
```

If the mapping doesn't include a column/index label, it isn't renamed. Note that extra labels in the mapping don't throw an error.

DataFrame.rename() also supports an "axis-style" calling convention, where you specify a single `mapper` and the `axis` to apply that mapping to.

```
In [247]: df.rename({"one": "foo", "two": "bar"}, axis="columns")
Out[247]:
        foo       bar     three
a  1.394981  1.772517       NaN
b  0.343054  1.912123 -0.050390
c  0.695246  1.478369  1.227435
d       NaN  0.279344 -0.613172

In [248]: df.rename({"a": "apple", "b": "banana", "d": "durian"}, axis="index")
Out[248]:
             one       two     three
apple   1.394981  1.772517       NaN
banana  0.343054  1.912123 -0.050390
c       0.695246  1.478369  1.227435
durian       NaN  0.279344 -0.613172
```

The *rename()* method also provides an `inplace` named parameter that is by default `False` and copies the underlying data. Pass `inplace=True` to rename the data in place.

Finally, *rename()* also accepts a scalar or list-like for altering the `Series.name` attribute.

```
In [249]: s.rename("scalar-name")
Out[249]:
a   -0.186646
b   -1.692424
c   -0.303893
d   -1.425662
e    1.114285
Name: scalar-name, dtype: float64
```

The methods *DataFrame.rename_axis()* and *Series.rename_axis()* allow specific names of a `MultiIndex` to be changed (as opposed to the labels).

```
In [250]: df = pd.DataFrame(
   .....:     {"x": [1, 2, 3, 4, 5, 6], "y": [10, 20, 30, 40, 50, 60]},
   .....:     index=pd.MultiIndex.from_product(
   .....:         [["a", "b", "c"], [1, 2]], names=["let", "num"]
   .....:     ),
   .....: )
   .....:

In [251]: df
Out[251]:
```

(continues on next page)

(continued from previous page)

```
         x   y
let num
a   1    1  10
    2    2  20
b   1    3  30
    2    4  40
c   1    5  50
    2    6  60

In [252]: df.rename_axis(index={"let": "abc"})
Out[252]:
         x   y
abc num
a   1    1  10
    2    2  20
b   1    3  30
    2    4  40
c   1    5  50
    2    6  60

In [253]: df.rename_axis(index=str.upper)
Out[253]:
         x   y
LET NUM
a   1    1  10
    2    2  20
b   1    3  30
    2    4  40
c   1    5  50
    2    6  60
```

2.3.8 Iteration

The behavior of basic iteration over pandas objects depends on the type. When iterating over a Series, it is regarded as array-like, and basic iteration produces the values. DataFrames follow the dict-like convention of iterating over the "keys" of the objects.

In short, basic iteration (`for i in object`) produces:

- **Series**: values
- **DataFrame**: column labels

Thus, for example, iterating over a DataFrame gives you the column names:

```
In [254]: df = pd.DataFrame(
   .....:     {"col1": np.random.randn(3), "col2": np.random.randn(3)}, index=["a", "b",
↪"c"]
   .....: )
   .....:

In [255]: for col in df:
   .....:     print(col)
```

(continues on next page)

(continued from previous page)

```
   .....:
col1
col2
```

pandas objects also have the dict-like *items()* method to iterate over the (key, value) pairs.

To iterate over the rows of a DataFrame, you can use the following methods:

- *iterrows()*: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- *itertuples()*: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than *iterrows()*, and is in most cases preferable to use to iterate over the values of a DataFrame.

Warning: Iterating through pandas objects is generally **slow**. In many cases, iterating manually over the rows is not needed and can be avoided with one of the following approaches:

- Look for a *vectorized* solution: many operations can be performed using built-in methods or NumPy functions, (boolean) indexing, …
- When you have a function that cannot work on the full DataFrame/Series at once, it is better to use *apply()* instead of iterating over the values. See the docs on *function application*.
- If you need to do iterative manipulations on the values but performance is important, consider writing the inner loop with cython or numba. See the *enhancing performance* section for some examples of this approach.

Warning: You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect!

For example, in the following case setting the value has no effect:

```
In [256]: df = pd.DataFrame({"a": [1, 2, 3], "b": ["a", "b", "c"]})

In [257]: for index, row in df.iterrows():
   .....:     row["a"] = 10
   .....:

In [258]: df
Out[258]:
   a  b
0  1  a
1  2  b
2  3  c
```

items

Consistent with the dict-like interface, *items()* iterates through key-value pairs:

- **Series**: (index, scalar value) pairs
- **DataFrame**: (column, Series) pairs

For example:

```
In [259]: for label, ser in df.items():
   .....:     print(label)
   .....:     print(ser)
   .....:
a
0    1
1    2
2    3
Name: a, dtype: int64
b
0    a
1    b
2    c
Name: b, dtype: object
```

iterrows

iterrows() allows you to iterate through the rows of a DataFrame as Series objects. It returns an iterator yielding each index value along with a Series containing the data in each row:

```
In [260]: for row_index, row in df.iterrows():
   .....:     print(row_index, row, sep="\n")
   .....:
0
a    1
b    a
Name: 0, dtype: object
1
a    2
b    b
Name: 1, dtype: object
2
a    3
b    c
Name: 2, dtype: object
```

Note: Because *iterrows()* returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
In [261]: df_orig = pd.DataFrame([[1, 1.5]], columns=["int", "float"])

In [262]: df_orig.dtypes
Out[262]:
```

(continues on next page)

(continued from previous page)

```
int        int64
float    float64
dtype: object

In [263]: row = next(df_orig.iterrows())[1]

In [264]: row
Out[264]:
int      1.0
float    1.5
Name: 0, dtype: float64
```

All values in `row`, returned as a Series, are now upcasted to floats, also the original integer value in column `x`:

```
In [265]: row["int"].dtype
Out[265]: dtype('float64')

In [266]: df_orig["int"].dtype
Out[266]: dtype('int64')
```

To preserve dtypes while iterating over the rows, it is better to use *itertuples()* which returns namedtuples of the values and which is generally much faster than *iterrows()*.

For instance, a contrived way to transpose the DataFrame would be:

```
In [267]: df2 = pd.DataFrame({"x": [1, 2, 3], "y": [4, 5, 6]})

In [268]: print(df2)
   x  y
0  1  4
1  2  5
2  3  6

In [269]: print(df2.T)
   0  1  2
x  1  2  3
y  4  5  6

In [270]: df2_t = pd.DataFrame({idx: values for idx, values in df2.iterrows()})

In [271]: print(df2_t)
   0  1  2
x  1  2  3
y  4  5  6
```

itertuples

The `itertuples()` method will return an iterator yielding a namedtuple for each row in the DataFrame. The first element of the tuple will be the row's corresponding index value, while the remaining values are the row values.

For instance:

```
In [272]: for row in df.itertuples():
   .....:     print(row)
   .....:
Pandas(Index=0, a=1, b='a')
Pandas(Index=1, a=2, b='b')
Pandas(Index=2, a=3, b='c')
```

This method does not convert the row to a Series object; it merely returns the values inside a namedtuple. Therefore, `itertuples()` preserves the data type of the values and is generally faster as `iterrows()`.

Note: The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

2.3.9 .dt accessor

`Series` has an accessor to succinctly return datetime like properties for the *values* of the Series, if it is a datetime/period like Series. This will return a Series, indexed like the existing Series.

```
# datetime
In [273]: s = pd.Series(pd.date_range("20130101 09:10:12", periods=4))

In [274]: s
Out[274]:
0   2013-01-01 09:10:12
1   2013-01-02 09:10:12
2   2013-01-03 09:10:12
3   2013-01-04 09:10:12
dtype: datetime64[ns]

In [275]: s.dt.hour
Out[275]:
0    9
1    9
2    9
3    9
dtype: int64

In [276]: s.dt.second
Out[276]:
0    12
1    12
2    12
3    12
dtype: int64
```

(continues on next page)

(continued from previous page)

```
In [277]: s.dt.day
Out[277]:
0    1
1    2
2    3
3    4
dtype: int64
```

This enables nice expressions like this:

```
In [278]: s[s.dt.day == 2]
Out[278]:
1   2013-01-02 09:10:12
dtype: datetime64[ns]
```

You can easily produces tz aware transformations:

```
In [279]: stz = s.dt.tz_localize("US/Eastern")

In [280]: stz
Out[280]:
0   2013-01-01 09:10:12-05:00
1   2013-01-02 09:10:12-05:00
2   2013-01-03 09:10:12-05:00
3   2013-01-04 09:10:12-05:00
dtype: datetime64[ns, US/Eastern]

In [281]: stz.dt.tz
Out[281]: <DstTzInfo 'US/Eastern' LMT-1 day, 19:04:00 STD>
```

You can also chain these types of operations:

```
In [282]: s.dt.tz_localize("UTC").dt.tz_convert("US/Eastern")
Out[282]:
0   2013-01-01 04:10:12-05:00
1   2013-01-02 04:10:12-05:00
2   2013-01-03 04:10:12-05:00
3   2013-01-04 04:10:12-05:00
dtype: datetime64[ns, US/Eastern]
```

You can also format datetime values as strings with *Series.dt.strftime()* which supports the same format as the standard `strftime()`.

```
# DatetimeIndex
In [283]: s = pd.Series(pd.date_range("20130101", periods=4))

In [284]: s
Out[284]:
0   2013-01-01
1   2013-01-02
2   2013-01-03
3   2013-01-04
dtype: datetime64[ns]
```

(continues on next page)

(continued from previous page)

```
In [285]: s.dt.strftime("%Y/%m/%d")
Out[285]:
0    2013/01/01
1    2013/01/02
2    2013/01/03
3    2013/01/04
dtype: object
```

```
# PeriodIndex
In [286]: s = pd.Series(pd.period_range("20130101", periods=4))

In [287]: s
Out[287]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: period[D]

In [288]: s.dt.strftime("%Y/%m/%d")
Out[288]:
0    2013/01/01
1    2013/01/02
2    2013/01/03
3    2013/01/04
dtype: object
```

The `.dt` accessor works for period and timedelta dtypes.

```
# period
In [289]: s = pd.Series(pd.period_range("20130101", periods=4, freq="D"))

In [290]: s
Out[290]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: period[D]

In [291]: s.dt.year
Out[291]:
0    2013
1    2013
2    2013
3    2013
dtype: int64

In [292]: s.dt.day
Out[292]:
0    1
```

(continues on next page)

(continued from previous page)

```
1    2
2    3
3    4
dtype: int64
```

```
# timedelta
In [293]: s = pd.Series(pd.timedelta_range("1 day 00:00:05", periods=4, freq="s"))

In [294]: s
Out[294]: 
0   1 days 00:00:05
1   1 days 00:00:06
2   1 days 00:00:07
3   1 days 00:00:08
dtype: timedelta64[ns]

In [295]: s.dt.days
Out[295]: 
0    1
1    1
2    1
3    1
dtype: int64

In [296]: s.dt.seconds
Out[296]: 
0    5
1    6
2    7
3    8
dtype: int64

In [297]: s.dt.components
Out[297]: 
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0     1      0        0        5             0             0            0
1     1      0        0        6             0             0            0
2     1      0        0        7             0             0            0
3     1      0        0        8             0             0            0
```

Note: `Series.dt` will raise a `TypeError` if you access with a non-datetime-like values.

2.3.10 Vectorized string methods

Series is equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the Series's str attribute and generally have names matching the equivalent (scalar) built-in string methods. For example:

```
In [298]: s = pd.Series(
   .....:     ["A", "B", "C", "Aaba", "Baca", np.nan, "CABA", "dog", "cat"],␣
↪dtype="string"
   .....: )
   .....:

In [299]: s.str.lower()
Out[299]:
0       a
1       b
2       c
3    aaba
4    baca
5    <NA>
6    caba
7     dog
8     cat
dtype: string
```

Powerful pattern-matching methods are provided as well, but note that pattern-matching generally uses regular expressions by default (and in some cases always uses them).

Note: Prior to pandas 1.0, string methods were only available on object -dtype Series. pandas 1.0 added the *StringDtype* which is dedicated to strings. See *Text data types* for more.

Please see *Vectorized String Methods* for a complete description.

2.3.11 Sorting

pandas supports three kinds of sorting: sorting by index labels, sorting by column values, and sorting by a combination of both.

By index

The *Series.sort_index()* and *DataFrame.sort_index()* methods are used to sort a pandas object by its index levels.

```
In [300]: df = pd.DataFrame(
   .....:     {
   .....:         "one": pd.Series(np.random.randn(3), index=["a", "b", "c"]),
   .....:         "two": pd.Series(np.random.randn(4), index=["a", "b", "c", "d"]),
   .....:         "three": pd.Series(np.random.randn(3), index=["b", "c", "d"]),
   .....:     }
   .....: )
   .....:
```

(continues on next page)

(continued from previous page)

```
In [301]: unsorted_df = df.reindex(
   .....:     index=["a", "d", "c", "b"], columns=["three", "two", "one"]
   .....: )
   .....:

In [302]: unsorted_df
Out[302]:
      three       two       one
a       NaN -1.152244  0.562973
d -0.252916 -0.109597       NaN
c  1.273388 -0.167123  0.640382
b -0.098217  0.009797 -1.299504

# DataFrame
In [303]: unsorted_df.sort_index()
Out[303]:
      three       two       one
a       NaN -1.152244  0.562973
b -0.098217  0.009797 -1.299504
c  1.273388 -0.167123  0.640382
d -0.252916 -0.109597       NaN

In [304]: unsorted_df.sort_index(ascending=False)
Out[304]:
      three       two       one
d -0.252916 -0.109597       NaN
c  1.273388 -0.167123  0.640382
b -0.098217  0.009797 -1.299504
a       NaN -1.152244  0.562973

In [305]: unsorted_df.sort_index(axis=1)
Out[305]:
        one     three       two
a  0.562973       NaN -1.152244
d       NaN -0.252916 -0.109597
c  0.640382  1.273388 -0.167123
b -1.299504 -0.098217  0.009797

# Series
In [306]: unsorted_df["three"].sort_index()
Out[306]:
a         NaN
b   -0.098217
c    1.273388
d   -0.252916
Name: three, dtype: float64
```

New in version 1.1.0.

Sorting by index also supports a `key` parameter that takes a callable function to apply to the index being sorted. For `MultiIndex` objects, the key is applied per-level to the levels specified by `level`.

```
In [307]: s1 = pd.DataFrame({"a": ["B", "a", "C"], "b": [1, 2, 3], "c": [2, 3, 4]}).set_
↪index(
   .....:     list("ab")
   .....: )
   .....:

In [308]: s1
Out[308]: 
     c
a b   
B 1  2
a 2  3
C 3  4
```

```
In [309]: s1.sort_index(level="a")
Out[309]: 
     c
a b   
B 1  2
C 3  4
a 2  3

In [310]: s1.sort_index(level="a", key=lambda idx: idx.str.lower())
Out[310]: 
     c
a b   
a 2  3
B 1  2
C 3  4
```

For information on key sorting by value, see *value sorting*.

By values

The *Series.sort_values()* method is used to sort a Series by its values. The *DataFrame.sort_values()* method is used to sort a DataFrame by its column or row values. The optional by parameter to *DataFrame.sort_values()* may used to specify one or more columns to use to determine the sorted order.

```
In [311]: df1 = pd.DataFrame(
   .....:     {"one": [2, 1, 1, 1], "two": [1, 3, 2, 4], "three": [5, 4, 3, 2]}
   .....: )
   .....:

In [312]: df1.sort_values(by="two")
Out[312]: 
   one  two  three
0    2    1      5
2    1    2      3
1    1    3      4
3    1    4      2
```

The by parameter can take a list of column names, e.g.:

```
In [313]: df1[["one", "two", "three"]].sort_values(by=["one", "two"])
Out[313]:
   one  two  three
2    1    2      3
1    1    3      4
3    1    4      2
0    2    1      5
```

These methods have special treatment of NA values via the `na_position` argument:

```
In [314]: s[2] = np.nan

In [315]: s.sort_values()
Out[315]:
0       A
3    Aaba
1       B
4    Baca
6    CABA
8     cat
7     dog
2    <NA>
5    <NA>
dtype: string

In [316]: s.sort_values(na_position="first")
Out[316]:
2    <NA>
5    <NA>
0       A
3    Aaba
1       B
4    Baca
6    CABA
8     cat
7     dog
dtype: string
```

New in version 1.1.0.

Sorting also supports a `key` parameter that takes a callable function to apply to the values being sorted.

```
In [317]: s1 = pd.Series(["B", "a", "C"])
```

```
In [318]: s1.sort_values()
Out[318]:
0    B
2    C
1    a
dtype: object

In [319]: s1.sort_values(key=lambda x: x.str.lower())
Out[319]:
```

(continues on next page)

(continued from previous page)

```
1    a
0    B
2    C
dtype: object
```

key will be given the *Series* of values and should return a Series or array of the same shape with the transformed values. For DataFrame objects, the key is applied per column, so the key should still expect a Series and return a Series, e.g.

```
In [320]: df = pd.DataFrame({"a": ["B", "a", "C"], "b": [1, 2, 3]})
```

```
In [321]: df.sort_values(by="a")
Out[321]:
   a  b
0  B  1
2  C  3
1  a  2

In [322]: df.sort_values(by="a", key=lambda col: col.str.lower())
Out[322]:
   a  b
1  a  2
0  B  1
2  C  3
```

The name or type of each column can be used to apply different functions to different columns.

By indexes and values

Strings passed as the by parameter to *DataFrame.sort_values()* may refer to either columns or index level names.

```
# Build MultiIndex
In [323]: idx = pd.MultiIndex.from_tuples(
   .....:     [("a", 1), ("a", 2), ("a", 2), ("b", 2), ("b", 1), ("b", 1)]
   .....: )
   .....:

In [324]: idx.names = ["first", "second"]

# Build DataFrame
In [325]: df_multi = pd.DataFrame({"A": np.arange(6, 0, -1)}, index=idx)

In [326]: df_multi
Out[326]:
              A
first second
a     1       6
      2       5
      2       4
b     2       3
      1       2
      1       1
```

Sort by 'second' (index) and 'A' (column)

```
In [327]: df_multi.sort_values(by=["second", "A"])
Out[327]:
              A
first second
b     1       1
      1       2
a     1       6
b     2       3
a     2       4
      2       5
```

Note: If a string matches both a column name and an index level name then a warning is issued and the column takes precedence. This will result in an ambiguity error in a future version.

searchsorted

Series has the *searchsorted()* method, which works similarly to `numpy.ndarray.searchsorted()`.

```
In [328]: ser = pd.Series([1, 2, 3])

In [329]: ser.searchsorted([0, 3])
Out[329]: array([0, 2])

In [330]: ser.searchsorted([0, 4])
Out[330]: array([0, 3])

In [331]: ser.searchsorted([1, 3], side="right")
Out[331]: array([1, 3])

In [332]: ser.searchsorted([1, 3], side="left")
Out[332]: array([0, 2])

In [333]: ser = pd.Series([3, 1, 2])

In [334]: ser.searchsorted([0, 3], sorter=np.argsort(ser))
Out[334]: array([0, 2])
```

smallest / largest values

`Series` has the *nsmallest()* and *nlargest()* methods which return the smallest or largest n values. For a large `Series` this can be much faster than sorting the entire Series and calling `head(n)` on the result.

```
In [335]: s = pd.Series(np.random.permutation(10))

In [336]: s
Out[336]:
0    2
1    0
```

(continues on next page)

(continued from previous page)

```
2    3
3    7
4    1
5    5
6    9
7    6
8    8
9    4
dtype: int64

In [337]: s.sort_values()
Out[337]: 
1    0
4    1
0    2
2    3
9    4
5    5
7    6
3    7
8    8
6    9
dtype: int64

In [338]: s.nsmallest(3)
Out[338]: 
1    0
4    1
0    2
dtype: int64

In [339]: s.nlargest(3)
Out[339]: 
6    9
8    8
3    7
dtype: int64
```

DataFrame also has the nlargest and nsmallest methods.

```
In [340]: df = pd.DataFrame(
   .....:     {
   .....:         "a": [-2, -1, 1, 10, 8, 11, -1],
   .....:         "b": list("abdceff"),
   .....:         "c": [1.0, 2.0, 4.0, 3.2, np.nan, 3.0, 4.0],
   .....:     }
   .....: )
   .....: 

In [341]: df.nlargest(3, "a")
Out[341]: 
    a  b    c
```

(continues on next page)

(continued from previous page)

```
5  11  f  3.0
3  10  c  3.2
4   8  e  NaN

In [342]: df.nlargest(5, ["a", "c"])
Out[342]:
    a  b    c
5  11  f  3.0
3  10  c  3.2
4   8  e  NaN
2   1  d  4.0
6  -1  f  4.0

In [343]: df.nsmallest(3, "a")
Out[343]:
   a  b    c
0 -2  a  1.0
1 -1  b  2.0
6 -1  f  4.0

In [344]: df.nsmallest(5, ["a", "c"])
Out[344]:
   a  b    c
0 -2  a  1.0
1 -1  b  2.0
6 -1  f  4.0
2  1  d  4.0
4  8  e  NaN
```

Sorting by a MultiIndex column

You must be explicit about sorting when the column is a MultiIndex, and fully specify all levels to by.

```
In [345]: df1.columns = pd.MultiIndex.from_tuples(
   .....:     [("a", "one"), ("a", "two"), ("b", "three")]
   .....: )
   .....:

In [346]: df1.sort_values(by=("a", "two"))
Out[346]:
    a         b
  one two three
0   2   1     5
2   1   2     3
1   1   3     4
3   1   4     2
```

2.3.12 Copying

The `copy()` method on pandas objects copies the underlying data (though not the axis indexes, since they are immutable) and returns a new object. Note that **it is seldom necessary to copy objects**. For example, there are only a handful of ways to alter a DataFrame *in-place*:

- Inserting, deleting, or modifying a column.
- Assigning to the `index` or `columns` attributes.
- For homogeneous data, directly modifying the values via the `values` attribute or advanced indexing.

To be clear, no pandas method has the side effect of modifying your data; almost every method returns a new object, leaving the original object untouched. If the data is modified, it is because you did so explicitly.

2.3.13 dtypes

For the most part, pandas uses NumPy arrays and dtypes for Series or individual columns of a DataFrame. NumPy provides support for `float`, `int`, `bool`, `timedelta64[ns]` and `datetime64[ns]` (note that NumPy does not support timezone-aware datetimes).

pandas and third-party libraries *extend* NumPy's type system in a few places. This section describes the extensions pandas has made internally. See *Extension types* for how to write your own extension that works with pandas. See ecosystem.extensions for a list of third-party libraries that have implemented an extension.

The following table lists all of pandas extension types. For methods requiring `dtype` arguments, strings can be specified as indicated. See the respective documentation sections for more on each type.

Kind of Data	Data Type	Scalar	Array	String Aliases
tz-aware datetime	`DatetimeTZDtype`	`Timestamp`	`arrays.DatetimeArray`	`'datetime64[ns, <tz>]'`
Categorical	`CategoricalDtype`	(none)	`Categorical`	`'category'`
period (time spans)	`PeriodDtype`	`Period`	`arrays.PeriodArray` `'Period[<freq>]'`	`'period[<freq>]'`,
sparse	`SparseDtype`	(none)	`arrays.SparseArray`	`'Sparse'`, `'Sparse[int]'`, `'Sparse[float]'`
intervals	`IntervalDtype`	`Interval`	`arrays.IntervalArray`	`'interval'`, `'Interval'`, `'Interval[<numpy_dtype>]'`, `'Interval[datetime64[ns, <tz>]]'`, `'Interval[timedelta64[<freq>]]'`
nullable integer	`Int64Dtype`, ...	(none)	`arrays.IntegerArray`	`'Int8'`, `'Int16'`, `'Int32'`, `'Int64'`, `'UInt8'`, `'UInt16'`, `'UInt32'`, `'UInt64'`
Strings	`StringDtype`	`str`	`arrays.StringArray`	`'string'`
Boolean (with NA)	`BooleanDtype`	`bool`	`arrays.BooleanArray`	`'boolean'`

pandas has two ways to store strings.

1. `object` dtype, which can hold any Python object, including strings.
2. *StringDtype*, which is dedicated to strings.

Generally, we recommend using *StringDtype*. See *Text data types* for more.

Finally, arbitrary objects may be stored using the `object` dtype, but should be avoided to the extent possible (for performance and interoperability with other libraries and methods. See *object conversion*).

A convenient *dtypes* attribute for DataFrame returns a Series with the data type of each column.

```
In [347]: dft = pd.DataFrame(
   .....:     {
   .....:         "A": np.random.rand(3),
   .....:         "B": 1,
   .....:         "C": "foo",
   .....:         "D": pd.Timestamp("20010102"),
   .....:         "E": pd.Series([1.0] * 3).astype("float32"),
   .....:         "F": False,
   .....:         "G": pd.Series([1] * 3, dtype="int8"),
   .....:     }
   .....: )
   .....:

In [348]: dft
Out[348]:
          A  B    C          D    E      F  G
0  0.035962  1  foo 2001-01-02  1.0  False  1
1  0.701379  1  foo 2001-01-02  1.0  False  1
2  0.281885  1  foo 2001-01-02  1.0  False  1

In [349]: dft.dtypes
Out[349]:
A           float64
B             int64
C            object
D    datetime64[ns]
E           float32
F              bool
G              int8
dtype: object
```

On a `Series` object, use the *dtype* attribute.

```
In [350]: dft["A"].dtype
Out[350]: dtype('float64')
```

If a pandas object contains data with multiple dtypes *in a single column*, the dtype of the column will be chosen to accommodate all of the data types (`object` is the most general).

```
# these ints are coerced to floats
In [351]: pd.Series([1, 2, 3, 4, 5, 6.0])
Out[351]:
0    1.0
```

(continues on next page)

(continued from previous page)

```
1    2.0
2    3.0
3    4.0
4    5.0
5    6.0
dtype: float64

# string data forces an ``object`` dtype
In [352]: pd.Series([1, 2, 3, 6.0, "foo"])
Out[352]:
0      1
1      2
2      3
3    6.0
4    foo
dtype: object
```

The number of columns of each type in a `DataFrame` can be found by calling `DataFrame.dtypes.value_counts()`.

```
In [353]: dft.dtypes.value_counts()
Out[353]:
float64           1
int64             1
object            1
datetime64[ns]    1
float32           1
bool              1
int8              1
dtype: int64
```

Numeric dtypes will propagate and can coexist in DataFrames. If a dtype is passed (either directly via the `dtype` keyword, a passed `ndarray`, or a passed `Series`), then it will be preserved in DataFrame operations. Furthermore, different numeric dtypes will **NOT** be combined. The following example will give you a taste.

```
In [354]: df1 = pd.DataFrame(np.random.randn(8, 1), columns=["A"], dtype="float32")

In [355]: df1
Out[355]:
          A
0  0.224364
1  1.890546
2  0.182879
3  0.787847
4 -0.188449
5  0.667715
6 -0.011736
7 -0.399073

In [356]: df1.dtypes
Out[356]:
A    float32
dtype: object
```

(continues on next page)

(continued from previous page)

```
In [357]: df2 = pd.DataFrame(
   .....:     {
   .....:         "A": pd.Series(np.random.randn(8), dtype="float16"),
   .....:         "B": pd.Series(np.random.randn(8)),
   .....:         "C": pd.Series(np.array(np.random.randn(8), dtype="uint8")),
   .....:     }
   .....: )
   .....:

In [358]: df2
Out[358]:
          A         B    C
0  0.823242  0.256090    0
1  1.607422  1.426469    0
2 -0.333740 -0.416203  255
3 -0.063477  1.139976    0
4 -1.014648 -1.193477    0
5  0.678711  0.096706    0
6 -0.040863 -1.956850    1
7 -0.357422 -0.714337    0

In [359]: df2.dtypes
Out[359]:
A    float16
B    float64
C      uint8
dtype: object
```

defaults

By default integer types are `int64` and float types are `float64`, *regardless* of platform (32-bit or 64-bit). The following will all result in `int64` dtypes.

```
In [360]: pd.DataFrame([1, 2], columns=["a"]).dtypes
Out[360]:
a    int64
dtype: object

In [361]: pd.DataFrame({"a": [1, 2]}).dtypes
Out[361]:
a    int64
dtype: object

In [362]: pd.DataFrame({"a": 1}, index=list(range(2))).dtypes
Out[362]:
a    int64
dtype: object
```

Note that Numpy will choose *platform-dependent* types when creating arrays. The following **WILL** result in `int32` on 32-bit platform.

```
In [363]: frame = pd.DataFrame(np.array([1, 2]))
```

upcasting

Types can potentially be *upcasted* when combined with other types, meaning they are promoted from the current type (e.g. `int` to `float`).

```
In [364]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2

In [365]: df3
Out[365]:
          A         B      C
0  1.047606  0.256090    0.0
1  3.497968  1.426469    0.0
2 -0.150862 -0.416203  255.0
3  0.724370  1.139976    0.0
4 -1.203098 -1.193477    0.0
5  1.346426  0.096706    0.0
6 -0.052599 -1.956850    1.0
7 -0.756495 -0.714337    0.0

In [366]: df3.dtypes
Out[366]:
A    float32
B    float64
C    float64
dtype: object
```

`DataFrame.to_numpy()` will return the *lower-common-denominator* of the dtypes, meaning the dtype that can accommodate **ALL** of the types in the resulting homogeneous dtyped NumPy array. This can force some *upcasting*.

```
In [367]: df3.to_numpy().dtype
Out[367]: dtype('float64')
```

astype

You can use the `astype()` method to explicitly convert dtypes from one to another. These will by default return a copy, even if the dtype was unchanged (pass `copy=False` to change this behavior). In addition, they will raise an exception if the astype operation is invalid.

Upcasting is always according to the **NumPy** rules. If two different dtypes are involved in an operation, then the more *general* one will be used as the result of the operation.

```
In [368]: df3
Out[368]:
          A         B      C
0  1.047606  0.256090    0.0
1  3.497968  1.426469    0.0
2 -0.150862 -0.416203  255.0
3  0.724370  1.139976    0.0
4 -1.203098 -1.193477    0.0
```

(continues on next page)

(continued from previous page)

```
5  1.346426  0.096706    0.0
6 -0.052599 -1.956850    1.0
7 -0.756495 -0.714337    0.0

In [369]: df3.dtypes
Out[369]:
A    float32
B    float64
C    float64
dtype: object

# conversion of dtypes
In [370]: df3.astype("float32").dtypes
Out[370]:
A    float32
B    float32
C    float32
dtype: object
```

Convert a subset of columns to a specified type using *astype()*.

```
In [371]: dft = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6], "c": [7, 8, 9]})

In [372]: dft[["a", "b"]] = dft[["a", "b"]].astype(np.uint8)

In [373]: dft
Out[373]:
   a  b  c
0  1  4  7
1  2  5  8
2  3  6  9

In [374]: dft.dtypes
Out[374]:
a    uint8
b    uint8
c    int64
dtype: object
```

Convert certain columns to a specific dtype by passing a dict to *astype()*.

```
In [375]: dft1 = pd.DataFrame({"a": [1, 0, 1], "b": [4, 5, 6], "c": [7, 8, 9]})

In [376]: dft1 = dft1.astype({"a": np.bool_, "c": np.float64})

In [377]: dft1
Out[377]:
       a  b    c
0   True  4  7.0
1  False  5  8.0
2   True  6  9.0
```

(continues on next page)

(continued from previous page)

```
In [378]: dft1.dtypes
Out[378]:
a       bool
b      int64
c    float64
dtype: object
```

Note: When trying to convert a subset of columns to a specified type using *astype()* and *loc()*, upcasting occurs.

loc() tries to fit in what we are assigning to the current dtypes, while `[]` will overwrite them taking the dtype from the right hand side. Therefore the following piece of code produces the unintended result.

```
In [379]: dft = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6], "c": [7, 8, 9]})

In [380]: dft.loc[:, ["a", "b"]].astype(np.uint8).dtypes
Out[380]:
a    uint8
b    uint8
dtype: object

In [381]: dft.loc[:, ["a", "b"]] = dft.loc[:, ["a", "b"]].astype(np.uint8)

In [382]: dft.dtypes
Out[382]:
a    int64
b    int64
c    int64
dtype: object
```

object conversion

pandas offers various functions to try to force conversion of types from the `object` dtype to other types. In cases where the data is already of the correct type, but stored in an `object` array, the *DataFrame.infer_objects()* and *Series.infer_objects()* methods can be used to soft convert to the correct type.

```
In [383]: import datetime

In [384]: df = pd.DataFrame(
   .....:     [
   .....:         [1, 2],
   .....:         ["a", "b"],
   .....:         [datetime.datetime(2016, 3, 2), datetime.datetime(2016, 3,␣
↪2)],
   .....:     ]
   .....: )
   .....:

In [385]: df = df.T

In [386]: df
```

(continues on next page)

(continued from previous page)

```
Out[386]:
   0  1          2
0  1  a 2016-03-02
1  2  b 2016-03-02

In [387]: df.dtypes
Out[387]:
0             object
1             object
2    datetime64[ns]
dtype: object
```

Because the data was transposed the original inference stored all columns as object, which `infer_objects` will correct.

```
In [388]: df.infer_objects().dtypes
Out[388]:
0              int64
1             object
2    datetime64[ns]
dtype: object
```

The following functions are available for one dimensional object arrays or scalars to perform hard conversion of objects to a specified type:

- *to_numeric()* (conversion to numeric dtypes)

```
In [389]: m = ["1.1", 2, 3]

In [390]: pd.to_numeric(m)
Out[390]: array([1.1, 2. , 3. ])
```

- *to_datetime()* (conversion to datetime objects)

```
In [391]: import datetime

In [392]: m = ["2016-07-09", datetime.datetime(2016, 3, 2)]

In [393]: pd.to_datetime(m)
Out[393]: DatetimeIndex(['2016-07-09', '2016-03-02'], dtype='datetime64[ns]', 
↪freq=None)
```

- *to_timedelta()* (conversion to timedelta objects)

```
In [394]: m = ["5us", pd.Timedelta("1day")]

In [395]: pd.to_timedelta(m)
Out[395]: TimedeltaIndex(['0 days 00:00:00.000005', '1 days 00:00:00'], dtype=
↪'timedelta64[ns]', freq=None)
```

To force a conversion, we can pass in an `errors` argument, which specifies how pandas should deal with elements that cannot be converted to desired dtype or object. By default, `errors='raise'`, meaning that any errors encountered will be raised during the conversion process. However, if `errors='coerce'`, these errors will be ignored and pandas will convert problematic elements to `pd.NaT` (for datetime and timedelta) or `np.nan` (for numeric). This might be useful if you are reading in data which is mostly of the desired dtype (e.g. numeric, datetime), but occasionally has non-conforming elements intermixed that you want to represent as missing:

```
In [396]: import datetime

In [397]: m = ["apple", datetime.datetime(2016, 3, 2)]

In [398]: pd.to_datetime(m, errors="coerce")
Out[398]: DatetimeIndex(['NaT', '2016-03-02'], dtype='datetime64[ns]', freq=None)

In [399]: m = ["apple", 2, 3]

In [400]: pd.to_numeric(m, errors="coerce")
Out[400]: array([nan,  2.,  3.])

In [401]: m = ["apple", pd.Timedelta("1day")]

In [402]: pd.to_timedelta(m, errors="coerce")
Out[402]: TimedeltaIndex([NaT, '1 days'], dtype='timedelta64[ns]', freq=None)
```

The `errors` parameter has a third option of `errors='ignore'`, which will simply return the passed in data if it encounters any errors with the conversion to a desired data type:

```
In [403]: import datetime

In [404]: m = ["apple", datetime.datetime(2016, 3, 2)]

In [405]: pd.to_datetime(m, errors="ignore")
Out[405]: Index(['apple', 2016-03-02 00:00:00], dtype='object')

In [406]: m = ["apple", 2, 3]

In [407]: pd.to_numeric(m, errors="ignore")
Out[407]: array(['apple', 2, 3], dtype=object)

In [408]: m = ["apple", pd.Timedelta("1day")]

In [409]: pd.to_timedelta(m, errors="ignore")
Out[409]: array(['apple', Timedelta('1 days 00:00:00')], dtype=object)
```

In addition to object conversion, *to_numeric()* provides another argument `downcast`, which gives the option of downcasting the newly (or already) numeric data to a smaller dtype, which can conserve memory:

```
In [410]: m = ["1", 2, 3]

In [411]: pd.to_numeric(m, downcast="integer")  # smallest signed int dtype
Out[411]: array([1, 2, 3], dtype=int8)

In [412]: pd.to_numeric(m, downcast="signed")  # same as 'integer'
Out[412]: array([1, 2, 3], dtype=int8)

In [413]: pd.to_numeric(m, downcast="unsigned")  # smallest unsigned int dtype
Out[413]: array([1, 2, 3], dtype=uint8)
```

(continues on next page)

(continued from previous page)

```
In [414]: pd.to_numeric(m, downcast="float")  # smallest float dtype
Out[414]: array([1., 2., 3.], dtype=float32)
```

As these methods apply only to one-dimensional arrays, lists or scalars; they cannot be used directly on multi-dimensional objects such as DataFrames. However, with `apply()`, we can "apply" the function over each column efficiently:

```
In [415]: import datetime

In [416]: df = pd.DataFrame([["2016-07-09", datetime.datetime(2016, 3, 2)]] * 2, dtype="O
↪")

In [417]: df
Out[417]: 
            0                    1
0  2016-07-09  2016-03-02 00:00:00
1  2016-07-09  2016-03-02 00:00:00

In [418]: df.apply(pd.to_datetime)
Out[418]: 
           0          1
0 2016-07-09 2016-03-02
1 2016-07-09 2016-03-02

In [419]: df = pd.DataFrame([["1.1", 2, 3]] * 2, dtype="O")

In [420]: df
Out[420]: 
     0  1  2
0  1.1  2  3
1  1.1  2  3

In [421]: df.apply(pd.to_numeric)
Out[421]: 
     0  1  2
0  1.1  2  3
1  1.1  2  3

In [422]: df = pd.DataFrame([["5us", pd.Timedelta("1day")]] * 2, dtype="O")

In [423]: df
Out[423]: 
     0                1
0  5us  1 days 00:00:00
1  5us  1 days 00:00:00

In [424]: df.apply(pd.to_timedelta)
Out[424]: 
                       0      1
0 0 days 00:00:00.000005 1 days
1 0 days 00:00:00.000005 1 days
```

gotchas

Performing selection operations on `integer` type data can easily upcast the data to `floating`. The dtype of the input data will be preserved in cases where `nans` are not introduced. See also *Support for integer NA*.

```
In [425]: dfi = df3.astype("int32")

In [426]: dfi["E"] = 1

In [427]: dfi
Out[427]: 
   A  B    C  E
0  1  0    0  1
1  3  1    0  1
2  0  0  255  1
3  0  1    0  1
4 -1 -1    0  1
5  1  0    0  1
6  0 -1    1  1
7  0  0    0  1

In [428]: dfi.dtypes
Out[428]: 
A    int32
B    int32
C    int32
E    int64
dtype: object

In [429]: casted = dfi[dfi > 0]

In [430]: casted
Out[430]: 
     A    B      C  E
0  1.0  NaN    NaN  1
1  3.0  1.0    NaN  1
2  NaN  NaN  255.0  1
3  NaN  1.0    NaN  1
4  NaN  NaN    NaN  1
5  1.0  NaN    NaN  1
6  NaN  NaN    1.0  1
7  NaN  NaN    NaN  1

In [431]: casted.dtypes
Out[431]: 
A    float64
B    float64
C    float64
E      int64
dtype: object
```

While float dtypes are unchanged.

```
In [432]: dfa = df3.copy()

In [433]: dfa["A"] = dfa["A"].astype("float32")

In [434]: dfa.dtypes
Out[434]:
A    float32
B    float64
C    float64
dtype: object

In [435]: casted = dfa[df2 > 0]

In [436]: casted
Out[436]:
          A         B      C
0  1.047606  0.256090    NaN
1  3.497968  1.426469    NaN
2       NaN       NaN  255.0
3       NaN  1.139976    NaN
4       NaN       NaN    NaN
5  1.346426  0.096706    NaN
6       NaN       NaN    1.0
7       NaN       NaN    NaN

In [437]: casted.dtypes
Out[437]:
A    float32
B    float64
C    float64
dtype: object
```

2.3.14 Selecting columns based on `dtype`

The *select_dtypes()* method implements subsetting of columns based on their `dtype`.

First, let's create a *DataFrame* with a slew of different dtypes:

```
In [438]: df = pd.DataFrame(
   .....:     {
   .....:         "string": list("abc"),
   .....:         "int64": list(range(1, 4)),
   .....:         "uint8": np.arange(3, 6).astype("u1"),
   .....:         "float64": np.arange(4.0, 7.0),
   .....:         "bool1": [True, False, True],
   .....:         "bool2": [False, True, False],
   .....:         "dates": pd.date_range("now", periods=3),
   .....:         "category": pd.Series(list("ABC")).astype("category"),
   .....:     }
   .....: )
   .....:
```

(continues on next page)

(continued from previous page)

```
In [439]: df["tdeltas"] = df.dates.diff()

In [440]: df["uint64"] = np.arange(3, 6).astype("u8")

In [441]: df["other_dates"] = pd.date_range("20130101", periods=3)

In [442]: df["tz_aware_dates"] = pd.date_range("20130101", periods=3, tz="US/Eastern")

In [443]: df
Out[443]: 
  string  int64  uint8  float64  bool1  bool2                      dates category␣
↪tdeltas  uint64 other_dates            tz_aware_dates
0      a      1      3      4.0   True  False 2022-01-22 10:50:03.741897        A     ␣
↪NaT       3  2013-01-01 2013-01-01 00:00:00-05:00
1      b      2      4      5.0  False   True 2022-01-23 10:50:03.741897        B   1␣
↪days       4  2013-01-02 2013-01-02 00:00:00-05:00
2      c      3      5      6.0   True  False 2022-01-24 10:50:03.741897        C   1␣
↪days       5  2013-01-03 2013-01-03 00:00:00-05:00
```

And the dtypes:

```
In [444]: df.dtypes
Out[444]: 
string                                object
int64                                  int64
uint8                                  uint8
float64                              float64
bool1                                   bool
bool2                                   bool
dates                         datetime64[ns]
category                            category
tdeltas                      timedelta64[ns]
uint64                                uint64
other_dates                   datetime64[ns]
tz_aware_dates    datetime64[ns, US/Eastern]
dtype: object
```

select_dtypes() has two parameters `include` and `exclude` that allow you to say "give me the columns *with* these dtypes" (`include`) and/or "give the columns *without* these dtypes" (`exclude`).

For example, to select `bool` columns:

```
In [445]: df.select_dtypes(include=[bool])
Out[445]: 
   bool1  bool2
0   True  False
1  False   True
2   True  False
```

You can also pass the name of a dtype in the NumPy dtype hierarchy:

```
In [446]: df.select_dtypes(include=["bool"])
Out[446]: 
```

(continues on next page)

(continued from previous page)

```
   bool1  bool2
0   True  False
1  False   True
2   True  False
```

select_dtypes() also works with generic dtypes as well.

For example, to select all numeric and boolean columns while excluding unsigned integers:

```
In [447]: df.select_dtypes(include=["number", "bool"], exclude=["unsignedinteger"])
Out[447]: 
   int64  float64  bool1  bool2 tdeltas
0      1      4.0   True  False     NaT
1      2      5.0  False   True  1 days
2      3      6.0   True  False  1 days
```

To select string columns you must use the `object` dtype:

```
In [448]: df.select_dtypes(include=["object"])
Out[448]: 
  string
0      a
1      b
2      c
```

To see all the child dtypes of a generic `dtype` like `numpy.number` you can define a function that returns a tree of child dtypes:

```
In [449]: def subdtypes(dtype):
   .....:     subs = dtype.__subclasses__()
   .....:     if not subs:
   .....:         return dtype
   .....:     return [dtype, [subdtypes(dt) for dt in subs]]
   .....: 
```

All NumPy dtypes are subclasses of `numpy.generic`:

```
In [450]: subdtypes(np.generic)
Out[450]: 
[numpy.generic,
 [[numpy.number,
   [[numpy.integer,
     [[numpy.signedinteger,
       [numpy.int8,
        numpy.int16,
        numpy.int32,
        numpy.int64,
        numpy.longlong,
        numpy.timedelta64]],
      [numpy.unsignedinteger,
       [numpy.uint8,
        numpy.uint16,
        numpy.uint32,
```

(continues on next page)

(continued from previous page)

```
        numpy.uint64,
        numpy.ulonglong]]]],
    [numpy.inexact,
     [[numpy.floating,
       [numpy.float16, numpy.float32, numpy.float64, numpy.float128]],
      [numpy.complexfloating,
       [numpy.complex64, numpy.complex128, numpy.complex256]]]]]],
  [numpy.flexible,
   [[numpy.character, [numpy.bytes_, numpy.str_]],
    [numpy.void, [numpy.record]]]],
  numpy.bool_,
  numpy.datetime64,
  numpy.object_]]
```

Note: pandas also defines the types `category`, and `datetime64[ns, tz]`, which are not integrated into the normal NumPy hierarchy and won't show up with the above function.

2.4 IO tools (text, CSV, HDF5, . . .)

The pandas I/O API is a set of top level `reader` functions accessed like *`pandas.read_csv()`* that generally return a pandas object. The corresponding `writer` functions are object methods that are accessed like *`DataFrame.to_csv()`*. Below is a table containing available `readers` and `writers`.

Format Type	Data Description	Reader	Writer
text	CSV	*read_csv*	*to_csv*
text	Fixed-Width Text File	*read_fwf*	
text	JSON	*read_json*	*to_json*
text	HTML	*read_html*	*to_html*
text	LaTeX		*Styler.to_latex*
text	XML	*read_xml*	*to_xml*
text	Local clipboard	*read_clipboard*	*to_clipboard*
binary	MS Excel	*read_excel*	*to_excel*
binary	OpenDocument	*read_excel*	
binary	HDF5 Format	*read_hdf*	*to_hdf*
binary	Feather Format	*read_feather*	*to_feather*
binary	Parquet Format	*read_parquet*	*to_parquet*
binary	ORC Format	*read_orc*	
binary	Stata	*read_stata*	*to_stata*
binary	SAS	*read_sas*	
binary	SPSS	*read_spss*	
binary	Python Pickle Format	*read_pickle*	*to_pickle*
SQL	SQL	*read_sql*	*to_sql*
SQL	Google BigQuery	*read_gbq*	*to_gbq*

Here is an informal performance comparison for some of these IO methods.

Note: For examples that use the `StringIO` class, make sure you import it with `from io import StringIO` for

Python 3.

2.4.1 CSV & text files

The workhorse function for reading text files (a.k.a. flat files) is *read_csv()*. See the *cookbook* for some advanced strategies.

Parsing options

read_csv() accepts the following common arguments:

Basic

filepath_or_buffer [various] Either a path to a file (a `str`, `pathlib.Path`, or `py:py._path.local.LocalPath`), URL (including http, ftp, and S3 locations), or any object with a `read()` method (such as an open file or `StringIO`).

sep [str, defaults to `','` for *read_csv()*, `\t` for *read_table()*] Delimiter to use. If sep is `None`, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\\r\\t'`.

delimiter [str, default `None`] Alternative argument name for sep.

delim_whitespace [boolean, default False] Specifies whether or not whitespace (e.g. `' '` or `'\t'`) will be used as the delimiter. Equivalent to setting `sep='\s+'`. If this option is set to `True`, nothing should be passed in for the `delimiter` parameter.

Column and index locations and names

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

The header can be a list of ints that specify row locations for a MultiIndex on the columns e.g. `[0,1,3]`. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so header=0 denotes the first line of data rather than the first line of the file.

names [array-like, default `None`] List of column names to use. If file contains no header row, then you should explicitly pass `header=None`. Duplicates in this list are not allowed.

index_col [int, str, sequence of int / str, or False, optional, default `None`] Column(s) to use as the row labels of the `DataFrame`, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used.

Note: `index_col=False` can be used to force pandas to *not* use the first column as the index, e.g. when you have a malformed file with delimiters at the end of each line.

The default value of `None` instructs pandas to guess. If the number of fields in the column header row is equal to the number of fields in the body of the data file, then a default index is used. If it is larger, then the first columns are used as index so that the remaining number of fields in the body are equal to the number of fields in the header.

The first row after the header is used to determine the number of columns, which will go into the index. If the subsequent rows contain less columns than the first row, they are filled with `NaN`.

This can be avoided through `usecols`. This ensures that the columns are taken as is and the trailing data are ignored.

usecols [list-like or callable, default `None`] Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in `names` or inferred from the document header row(s). If `names` are given, the document header row(s) are not taken into account. For example, a valid list-like `usecols` parameter would be `[0, 1, 2]` or `['foo', 'bar', 'baz']`.

Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`. To instantiate a DataFrame from `data` with element order preserved use `pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]` for columns in `['foo', 'bar']` order or `pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]` for `['bar', 'foo']` order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True:

```
In [1]: import pandas as pd

In [2]: from io import StringIO

In [3]: data = "col1,col2,col3\na,b,1\na,b,2\nc,d,3"

In [4]: pd.read_csv(StringIO(data))
Out[4]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [5]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ["COL1", "COL3"])
Out[5]:
  col1  col3
0    a     1
1    a     2
2    c     3
```

Using this parameter results in much faster parsing time and lower memory usage when using the c engine. The Python engine loads the data first before deciding which columns to drop.

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

Deprecated since version 1.4.0: Append `.squeeze("columns")` to the call to `{func_name}` to squeeze the data.

prefix [str, default `None`] Prefix to add to column numbers when no header, e.g. 'X' for X0, X1, …

Deprecated since version 1.4.0: Use a list comprehension on the DataFrame's columns after calling `read_csv`.

```
In [6]: data = "col1,col2,col3\na,b,1"

In [7]: df = pd.read_csv(StringIO(data))
```

(continues on next page)

(continued from previous page)

```
In [8]: df.columns = [f"pre_{col}" for col in df.columns]

In [9]: df
Out[9]:
  pre_col1 pre_col2  pre_col3
0        a        b         1
```

mangle_dupe_cols [boolean, default `True`] Duplicate columns will be specified as 'X', 'X.1'…'X.N', rather than 'X'…'X'. Passing in `False` will cause data to be overwritten if there are duplicate names in the columns.

General parsing configuration

dtype [Type name or dict of column -> type, default `None`] Data type for data or columns. E.g. `{'a': np.float64, 'b': np.int32}` (unsupported with `engine='python'`). Use `str` or `object` together with suitable `na_values` settings to preserve and not interpret dtype.

engine [{`'c'`, `'python'`, `'pyarrow'`}] Parser engine to use. The C and pyarrow engines are faster, while the python engine is currently more feature-complete. Multithreading is currently only supported by the pyarrow engine.

New in version 1.4.0: The "pyarrow" engine was added as an *experimental* engine, and some features are unsupported, or may not work correctly, with this engine.

converters [dict, default `None`] Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

true_values [list, default `None`] Values to consider as `True`.

false_values [list, default `None`] Values to consider as `False`.

skipinitialspace [boolean, default `False`] Skip spaces after delimiter.

skiprows [list-like or integer, default `None`] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise:

```
In [10]: data = "col1,col2,col3\na,b,1\na,b,2\nc,d,3"

In [11]: pd.read_csv(StringIO(data))
Out[11]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [12]: pd.read_csv(StringIO(data), skiprows=lambda x: x % 2 != 0)
Out[12]:
  col1 col2  col3
0    a    b     2
```

skipfooter [int, default `0`] Number of lines at bottom of file to skip (unsupported with engine='c').

nrows [int, default `None`] Number of rows of file to read. Useful for reading pieces of large files.

low_memory [boolean, default `True`] Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set `False`, or specify the type with the `dtype` parameter. Note that the entire file is read into a single `DataFrame` regardless, use the `chunksize` or `iterator` parameter to return the data in chunks. (Only valid with C parser)

memory_map [boolean, default False] If a filepath is provided for `filepath_or_buffer`, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

NA and missing data handling

na_values [scalar, str, list-like, or dict, default `None`] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. See *na values const* below for a list of the values interpreted as NaN by default.

keep_default_na [boolean, default `True`] Whether or not to include the default NaN values when parsing the data. Depending on whether `na_values` is passed in, the behavior is as follows:

- If `keep_default_na` is `True`, and `na_values` are specified, `na_values` is appended to the default NaN values used for parsing.
- If `keep_default_na` is `True`, and `na_values` are not specified, only the default NaN values are used for parsing.
- If `keep_default_na` is `False`, and `na_values` are specified, only the NaN values specified `na_values` are used for parsing.
- If `keep_default_na` is `False`, and `na_values` are not specified, no strings will be parsed as NaN.

Note that if `na_filter` is passed in as `False`, the `keep_default_na` and `na_values` parameters will be ignored.

na_filter [boolean, default `True`] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing `na_filter=False` can improve the performance of reading a large file.

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

skip_blank_lines [boolean, default `True`] If `True`, skip over blank lines rather than interpreting as NaN values.

Datetime handling

parse_dates [boolean or list of ints or names or list of lists or dict, default `False`.]

- If `True` -> try parsing the index.
- If `[1, 2, 3]` -> try parsing columns 1, 2, 3 each as a separate date column.
- If `[[1, 3]]` -> combine columns 1 and 3 and parse as a single date column.
- If `{'foo': [1, 3]}` -> parse columns 1, 3 as date and call result 'foo'. A fast-path exists for iso8601-formatted dates.

infer_datetime_format [boolean, default `False`] If `True` and parse_dates is enabled for a column, attempt to infer the datetime format to speed up the processing.

keep_date_col [boolean, default `False`] If `True` and parse_dates specifies combining multiple columns then keep the original columns.

date_parser [function, default `None`] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. pandas will try to call date_parser in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by parse_dates) as arguments; 2) concatenate (row-wise) the string values from the columns defined by parse_dates into a single array and pass that; and 3) call date_parser once for each row using one or more strings (corresponding to the columns defined by parse_dates) as arguments.

dayfirst [boolean, default `False`] DD/MM format dates, international and European format.

cache_dates [boolean, default True] If True, use a cache of unique, converted dates to apply the datetime conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets.

New in version 0.25.0.

Iteration

iterator [boolean, default `False`] Return `TextFileReader` object for iteration or getting chunks with `get_chunk()`.

chunksize [int, default `None`] Return `TextFileReader` object for iteration. See *iterating and chunking* below.

Quoting, compression, and file format

compression [{`'infer'`, `'gzip'`, `'bz2'`, `'zip'`, `'xz'`, `'zstd'`, `None`, `dict`}, default `'infer'`] For on-the-fly decompression of on-disk data. If 'infer', then use gzip, bz2, zip, xz, or zstandard if `filepath_or_buffer` is path-like ending in '.gz', '.bz2', '.zip', '.xz', '.zst', respectively, and no decompression otherwise. If using 'zip', the ZIP file must contain only one data file to be read in. Set to `None` for no decompression. 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`. 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 1.1.0: dict option extended to support `gzip` and `bz2`.

Changed in version 1.2.0: Previous versions forwarded dict entries for 'gzip' to `gzip.open`.

thousands [str, default `None`] Thousands separator.

decimal [str, default `'.'`] Character to recognize as decimal point. E.g. use `','` for European data.

float_precision [string, default None] Specifies which converter the C engine should use for floating-point values. The options are `None` for the ordinary converter, `high` for the high-precision converter, and `round_trip` for the round-trip converter.

lineterminator [str (length 1), default `None`] Character to break file into lines. Only valid with C parser.

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

quoting [int or `csv.QUOTE_*` instance, default `0`] Control field quoting behavior per `csv.QUOTE_*` constants. Use one of `QUOTE_MINIMAL` (0), `QUOTE_ALL` (1), `QUOTE_NONNUMERIC` (2) or `QUOTE_NONE` (3).

doublequote [boolean, default `True`] When `quotechar` is specified and `quoting` is not `QUOTE_NONE`, indicate whether or not to interpret two consecutive `quotechar` elements **inside** a field as a single `quotechar` element.

escapechar [str (length 1), default `None`] One-character string used to escape delimiter when quoting is `QUOTE_NONE`.

comment [str, default `None`] Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter `header` but not by `skiprows`. For example, if `comment='#'`, parsing '#empty\na,b,c\n1,2,3' with `header=0` will result in 'a,b,c' being treated as the header.

encoding [str, default None] Encoding to use for UTF when reading/writing (e.g. 'utf-8'). List of Python standard encodings.

dialect [str or csv.Dialect instance, default None] If provided, this parameter will override values (default or not) for the following parameters: delimiter, doublequote, escapechar, skipinitialspace, quotechar, and quoting. If it is necessary to override values, a ParserWarning will be issued. See csv.Dialect documentation for more details.

Error handling

error_bad_lines [boolean, optional, default None] Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no DataFrame will be returned. If False, then these "bad lines" will dropped from the DataFrame that is returned. See *bad lines* below.

Deprecated since version 1.3.0: The on_bad_lines parameter should be used instead to specify behavior upon encountering a bad line instead.

warn_bad_lines [boolean, optional, default None] If error_bad_lines is False, and warn_bad_lines is True, a warning for each "bad line" will be output.

Deprecated since version 1.3.0: The on_bad_lines parameter should be used instead to specify behavior upon encountering a bad line instead.

on_bad_lines [('error', 'warn', 'skip'), default 'error'] Specifies what to do upon encountering a bad line (a line with too many fields). Allowed values are :

- 'error', raise an ParserError when a bad line is encountered.
- 'warn', print a warning when a bad line is encountered and skip that line.
- 'skip', skip bad lines without raising or warning when they are encountered.

New in version 1.3.0.

Specifying column data types

You can indicate the data type for the whole DataFrame or individual columns:

```
In [13]: import numpy as np

In [14]: data = "a,b,c,d\n1,2,3,4\n5,6,7,8\n9,10,11"

In [15]: print(data)
a,b,c,d
1,2,3,4
5,6,7,8
9,10,11

In [16]: df = pd.read_csv(StringIO(data), dtype=object)

In [17]: df
Out[17]:
   a   b   c    d
0  1   2   3    4
1  5   6   7    8
2  9  10  11  NaN
```

(continues on next page)

(continued from previous page)

```
In [18]: df["a"][0]
Out[18]: '1'

In [19]: df = pd.read_csv(StringIO(data), dtype={"b": object, "c": np.float64, "d":
↪"Int64"})

In [20]: df.dtypes
Out[20]:
a      int64
b     object
c    float64
d      Int64
dtype: object
```

Fortunately, pandas offers more than one way to ensure that your column(s) contain only one `dtype`. If you're unfamiliar with these concepts, you can see *here* to learn more about dtypes, and *here* to learn more about `object` conversion in pandas.

For instance, you can use the `converters` argument of *read_csv()*:

```
In [21]: data = "col_1\n1\n2\n'A'\n4.22"

In [22]: df = pd.read_csv(StringIO(data), converters={"col_1": str})

In [23]: df
Out[23]:
  col_1
0     1
1     2
2   'A'
3  4.22

In [24]: df["col_1"].apply(type).value_counts()
Out[24]:
<class 'str'>    4
Name: col_1, dtype: int64
```

Or you can use the *to_numeric()* function to coerce the dtypes after reading in the data,

```
In [25]: df2 = pd.read_csv(StringIO(data))

In [26]: df2["col_1"] = pd.to_numeric(df2["col_1"], errors="coerce")

In [27]: df2
Out[27]:
   col_1
0   1.00
1   2.00
2    NaN
3   4.22

In [28]: df2["col_1"].apply(type).value_counts()
```

(continues on next page)

(continued from previous page)

```
Out[28]:
<class 'float'>    4
Name: col_1, dtype: int64
```

which will convert all valid parsing to floats, leaving the invalid parsing as `NaN`.

Ultimately, how you deal with reading in columns containing mixed dtypes depends on your specific needs. In the case above, if you wanted to `NaN` out the data anomalies, then *to_numeric()* is probably your best option. However, if you wanted for all the data to be coerced, no matter the type, then using the `converters` argument of *read_csv()* would certainly be worth trying.

Note: In some cases, reading in abnormal data with columns containing mixed dtypes will result in an inconsistent dataset. If you rely on pandas to infer the dtypes of your columns, the parsing engine will go and infer the dtypes for different chunks of the data, rather than the whole dataset at once. Consequently, you can end up with column(s) with mixed dtypes. For example,

```
In [29]: col_1 = list(range(500000)) + ["a", "b"] + list(range(500000))

In [30]: df = pd.DataFrame({"col_1": col_1})

In [31]: df.to_csv("foo.csv")

In [32]: mixed_df = pd.read_csv("foo.csv")

In [33]: mixed_df["col_1"].apply(type).value_counts()
Out[33]:
<class 'int'>    737858
<class 'str'>    262144
Name: col_1, dtype: int64

In [34]: mixed_df["col_1"].dtype
Out[34]: dtype('O')
```

will result with `mixed_df` containing an `int` dtype for certain chunks of the column, and `str` for others due to the mixed dtypes from the data that was read in. It is important to note that the overall column will be marked with a `dtype` of `object`, which is used for columns with mixed dtypes.

Specifying categorical dtype

`Categorical` columns can be parsed directly by specifying `dtype='category'` or `dtype=CategoricalDtype(categories, ordered)`.

```
In [35]: data = "col1,col2,col3\na,b,1\na,b,2\nc,d,3"

In [36]: pd.read_csv(StringIO(data))
Out[36]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

```

(continues on next page)

(continued from previous page)

```
In [37]: pd.read_csv(StringIO(data)).dtypes
Out[37]:
col1    object
col2    object
col3     int64
dtype: object

In [38]: pd.read_csv(StringIO(data), dtype="category").dtypes
Out[38]:
col1    category
col2    category
col3    category
dtype: object
```

Individual columns can be parsed as a `Categorical` using a dict specification:

```
In [39]: pd.read_csv(StringIO(data), dtype={"col1": "category"}).dtypes
Out[39]:
col1    category
col2      object
col3       int64
dtype: object
```

Specifying `dtype='category'` will result in an unordered `Categorical` whose `categories` are the unique values observed in the data. For more control on the categories and order, create a `CategoricalDtype` ahead of time, and pass that for that column's `dtype`.

```
In [40]: from pandas.api.types import CategoricalDtype

In [41]: dtype = CategoricalDtype(["d", "c", "b", "a"], ordered=True)

In [42]: pd.read_csv(StringIO(data), dtype={"col1": dtype}).dtypes
Out[42]:
col1    category
col2      object
col3       int64
dtype: object
```

When using `dtype=CategoricalDtype`, "unexpected" values outside of `dtype.categories` are treated as missing values.

```
In [43]: dtype = CategoricalDtype(["a", "b", "d"])  # No 'c'

In [44]: pd.read_csv(StringIO(data), dtype={"col1": dtype}).col1
Out[44]:
0      a
1      a
2    NaN
Name: col1, dtype: category
Categories (3, object): ['a', 'b', 'd']
```

This matches the behavior of `Categorical.set_categories()`.

Note: With `dtype='category'`, the resulting categories will always be parsed as strings (object dtype). If the categories are numeric they can be converted using the *to_numeric()* function, or as appropriate, another converter such as *to_datetime()*.

When `dtype` is a `CategoricalDtype` with homogeneous `categories` (all numeric, all datetimes, etc.), the conversion is done automatically.

```
In [45]: df = pd.read_csv(StringIO(data), dtype="category")

In [46]: df.dtypes
Out[46]:
col1    category
col2    category
col3    category
dtype: object

In [47]: df["col3"]
Out[47]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, object): ['1', '2', '3']

In [48]: df["col3"].cat.categories = pd.to_numeric(df["col3"].cat.categories)

In [49]: df["col3"]
Out[49]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, int64): [1, 2, 3]
```

Naming and using columns

Handling column names

A file may or may not have a header row. pandas assumes the first row should be used as the column names:

```
In [50]: data = "a,b,c\n1,2,3\n4,5,6\n7,8,9"

In [51]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [52]: pd.read_csv(StringIO(data))
Out[52]:
   a  b  c
```

(continues on next page)

(continued from previous page)

```
0  1  2  3
1  4  5  6
2  7  8  9
```

By specifying the `names` argument in conjunction with `header` you can indicate other names to use and whether or not to throw away the header row (if any):

```
In [53]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [54]: pd.read_csv(StringIO(data), names=["foo", "bar", "baz"], header=0)
Out[54]:
   foo  bar  baz
0    1    2    3
1    4    5    6
2    7    8    9

In [55]: pd.read_csv(StringIO(data), names=["foo", "bar", "baz"], header=None)
Out[55]:
  foo bar baz
0   a   b   c
1   1   2   3
2   4   5   6
3   7   8   9
```

If the header is in a row other than the first, pass the row number to `header`. This will skip the preceding rows:

```
In [56]: data = "skip this skip it\na,b,c\n1,2,3\n4,5,6\n7,8,9"

In [57]: pd.read_csv(StringIO(data), header=1)
Out[57]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

Note: Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first non-blank line of the file, if column names are passed explicitly then the behavior is identical to `header=None`.

Duplicate names parsing

If the file or header contains duplicate names, pandas will by default distinguish between them so as to prevent overwriting data:

```
In [58]: data = "a,b,a\n0,1,2\n3,4,5"

In [59]: pd.read_csv(StringIO(data))
Out[59]:
   a  b  a.1
0  0  1    2
1  3  4    5
```

There is no more duplicate data because `mangle_dupe_cols=True` by default, which modifies a series of duplicate columns 'X', ..., 'X' to become 'X', 'X.1', ..., 'X.N'. If `mangle_dupe_cols=False`, duplicate data can arise:

```
In [2]: data = 'a,b,a\n0,1,2\n3,4,5'
In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
Out[3]:
   a  b  a
0  2  1  2
1  5  4  5
```

To prevent users from encountering this problem with duplicate data, a `ValueError` exception is raised if `mangle_dupe_cols != True`:

```
In [2]: data = 'a,b,a\n0,1,2\n3,4,5'
In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
...
ValueError: Setting mangle_dupe_cols=False is not supported yet
```

Filtering columns (usecols)

The `usecols` argument allows you to select any subset of the columns in a file, either using the column names, position numbers or a callable:

```
In [60]: data = "a,b,c,d\n1,2,3,foo\n4,5,6,bar\n7,8,9,baz"

In [61]: pd.read_csv(StringIO(data))
Out[61]:
   a  b  c    d
0  1  2  3  foo
1  4  5  6  bar
2  7  8  9  baz

In [62]: pd.read_csv(StringIO(data), usecols=["b", "d"])
Out[62]:
   b    d
0  2  foo
1  5  bar
2  8  baz
```

(continues on next page)