
Python for Data Science

Release 24.1.0

Veit Schiele

May 07, 2024

CONTENTS

1	Introduction	3
1.1	Target groups	3
1.2	Structure of the Python for Data Science tutorial	3
1.3	Status	4
1.4	Follow us	4
1.5	Pull-Requests	4
2	Workspace	5
2.1	IPython	5
2.2	Jupyter	42
2.3	NumPy	42
2.4	pandas	63
3	Read, persist and provide data	157
3.1	Open data	158
3.2	pandas IO tools	159
3.3	Serialisation formats	160
3.4	Intake	188
3.5	htpx	204
3.6	Overview	209
3.7	Geodata	210
3.8	PostgreSQL	211
3.9	NoSQL databases	240
3.10	Application Programming Interface (API)	247
3.11	Glossary	273
4	Data cleansing and validation	279
4.1	Overview	280
5	Visualise data	327
6	Performance	329
6.1	k-Means example	329
6.2	Performance measurements	331
6.3	Search for existing implementations	340
6.4	Find anti-patterns	340
6.5	Vectorisations with NumPy	342
6.6	Special data structures	342
6.7	Select compiler	344
6.8	Task planner	346
6.9	Multithreading, Multiprocessing and Async	350

7	Create a product	369
7.1	Manage code with Git	370
7.2	Manage data with DVC	469
7.3	Reproduce environments	480
7.4	Creating programme libraries and packages	514
7.5	Document	514
7.6	Licensing	515
7.7	Citing	523
7.8	Testing	535
7.9	Logging	535
7.10	Check and improve code quality and complexity	547
7.11	Security	567
8	Create web applications	573
9	Index	575
	Index	577

This is a tutorial on Data Science with Python. This immediately raises the question: What is Data Science? The term has become ubiquitous, but there is no single definition. Some even consider the term superfluous, because what science does not have to do with data? Nevertheless, it seems to me that data science is more than just hype: scientific data has become increasingly voluminous and often can no longer be adequately tapped with conventional mathematical and statistical methods alone – additional hacking skills are needed. However, it is not a new field of knowledge that you need to learn, but a set of skills that you can apply in your field. Whether you are analysing astronomical objects, analysing machines, forecasting stock prices or working with data in other fields, the goal of this tutorial is to enable you to solve tasks programmatically in your field.

This tutorial is not intended to be an introduction to Python or programming in general; for that there is the [Python basics](#) tutorial. Instead, it is intended to show the Python data science stack – libraries such as [IPython](#), [NumPy](#), [pandas](#), [Matplotlib](#) and related tools – so that you can subsequently effectively analyse and visualise your data.

INTRODUCTION

1.1 Target groups

The target groups are diverse, from data scientists to data engineers and analysts to systems engineers. Their skills and workflows are very different. However, one of the great strengths of Python for Data Science is that it allows these different experts to work closely together in cross-functional teams.

Data scientists

explore data with different parameters and summarise the results.

Data engineers

check the quality of the code and make it more robust, efficient and scalable.

Data analysts

use the code provided by data engineers to systematically analyse the data.

System engineers

provide the research platform based on the [JupyterHub](#) on which the other roles can perform their work.

In this tutorial we address system engineers who want to build and run a platform based on Jupyter notebooks. We then explain how this platform can be used effectively by data scientists, data engineers and analysts.

1.2 Structure of the Python for Data Science tutorial

From Chapter 2, the tutorial follows the prototype of a research project:

2. *Workspace* with the installation and configuration of *IPython*, *Jupyter notebooks* with *nbextensions* and *ipywidgets*.
3. *Read, persist and provide data* either through a *REST API* or directly from an *HTML page*.
4. *Data cleansing and validation* is a recurring task that involves removing or changing redundant, inconsistent or incorrectly formatted data.
5. *Visualise data* has been moved to a separate tutorial with the many different possibilities.
6. *Performance* introduces ways to make your code run faster.
7. *Create a product* shows what is necessary to achieve reproducible results: not only *reproducible environments* are needed, but also versioning of the *source code* and *data*. The source code should be *packed into programme libraries* with *documentation*, *licence(s)*, *tests* and *logging*. Finally, the chapter includes advice on *improving code quality* and *secure operation*.
8. *Create web applications* can either generate dashboards from Jupyter notebooks or require more comprehensive application logic, such as demonstrated in *Bokeh-Plots in Flask einbinden*, or provide data via a *RESTful API*.

:

1.3 Status

:

1.4 Follow us

- [GitHub](#)
- [Mastodon](#)

1.5 Pull-Requests

If you have suggestions for improvements and additions, I recommend that you create a [Fork](#) of my [GitHub Repository](#) and make your changes there. . You are also welcome to make a *pull request*. If the changes contained therein are small and atomic, I'll be happy to look at your suggestions.

The following guidelines help us to maintain the German translation of the tutorial:

- Write commit messages in English
- Start commit messages with a [Gitmoji](#)
- Stick to English names of files and folders.

WORKSPACE

Setting up the workspace includes installing and configuring *IPython* and *Jupyter* with *nbextensions* and *ipywidgets*, and *NumPy*.

2.1 IPython

IPython, or *Interactive Python*, was initially an advanced Python interpreter that has now grown into an extensive project designed to provide tools for the entire life cycle of research computing. Today, *IPython* is not only an interactive interface to Python, but also offers a number of useful syntactic additions for the language. In addition, *IPython* is closely related to the *Jupyter* project.

See also:

- Miki Tebeka - *IPython: The Productivity Booster*

2.1.1 Start the IPython shell

You can easily start *IPython* in a console:

```
$ pipenv run ipython
Python 3.7.0 (default, Aug 22 2018, 15:22:29)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Alternatively, you can use *IPython* in a *Jupyter* notebook. To do this, start the notebook server first:

```
$ pipenv run jupyter notebook
[I 17:35:02.419 NotebookApp] Serving notebooks from local directory: /Users/veit/cusy/
↳trn/Python4DataScience
[I 17:35:02.419 NotebookApp] The Jupyter Notebook is running at:
[I 17:35:02.427 NotebookApp] http://localhost:8888/?
↳token=72209334c2e325a68115902a63bd064db436c0c84aeced7f
[I 17:35:02.428 NotebookApp] Use Control-C to stop this server and shut down all kernels.
↳(twice to skip confirmation).
[C 17:35:02.497 NotebookApp]
```

The standard browser should then be opened with the specified URL. Often this is `http://localhost:8888`.

Now you can start a Python process in the browser by creating a new notebook.

2.1.2 IPython examples

Running Python code

Show Python version

```
[1]: import sys

sys.version_info

[1]: sys.version_info(major=3, minor=11, micro=4, releaselevel='final', serial=0)
```

Show versions of Python packages

Most Python packages provide a `__version__` method for this:

```
[2]: import pandas as pd

pd.__version__

[2]: '2.0.3'
```

Alternatively, you can use `version` from `importlib_metadata`:

```
[3]: from importlib_metadata import version

print(version("pandas"))

2.0.3
```

Information about the host operating system and the versions of installed Python packages

```
[4]: pd.show_versions()

/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/_
↳ distutils_hack/__init__.py:33: UserWarning: Setuptools is replacing distutils.
  warnings.warn("Setuptools is replacing distutils.")

INSTALLED VERSIONS
-----
commit           : 0f437949513225922d851e9581723d82120684a6
python           : 3.11.4.final.0
python-bits      : 64
OS               : Darwin
OS-release       : 22.5.0
Version          : Darwin Kernel Version 22.5.0: Thu Jun  8 22:22:23 PDT 2023; root:xnu-
↳ 8796.121.3~7/RELEASE_ARM64_T6020
machine         : arm64
```

(continues on next page)

(continued from previous page)

```

processor      : arm
byteorder     : little
LC_ALL        : None
LANG          : de_DE.UTF-8
LOCALE        : de_DE.UTF-8

pandas        : 2.0.3
numpy         : 1.23.5
pytz          : 2023.3
dateutil      : 2.8.2
setuptools    : 68.0.0
pip           : 23.1.2
Cython        : None
pytest        : 7.4.0
hypothesis    : 6.81.1
sphinx        : 7.0.1
blosc         : None
feather       : None
xlsxwriter    : None
lxml.etree    : 4.9.3
html5lib      : None
pymysql       : None
psycopg2      : None
jinja2        : 3.1.2
IPython       : 8.14.0
pandas_datareader: None
bs4           : 4.12.2
bottleneck    : None
brotli        : None
fastparquet   : 2023.7.0
fsspec        : 2023.6.0
gcsfs         : 2023.6.0
matplotlib    : 3.7.2
numba         : 0.57.1
numexpr       : None
odfpy         : None
openpyxl      : None
pandas_gbq    : None
pyarrow       : 12.0.1
pyreadstat    : None
pyxlsb        : None
s3fs          : 2023.6.0
scipy         : 1.11.1
snappy        : None
sqlalchemy    : None
tables        : None
tabulate      : 0.9.0
xarray        : 2023.6.0
xlrd          : None
zstandard     : None
tzdata        : 2023.3
qtpy          : None

```

(continues on next page)

(continued from previous page)

```
pyqt5          : None
```

Only use Python versions 3.8

```
[5]: import sys

assert sys.version_info[:2] >= (3, 8)
```

Shell commands

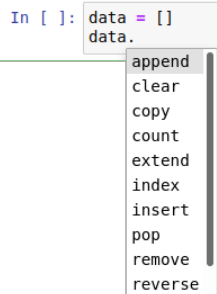
```
[6]: !python3 -V
Python 3.11.4
```

```
[7]: !python3 -m pip --version
pip 23.1.2 from /Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/
site-packages/pip (python 3.11)
```

Tab completion

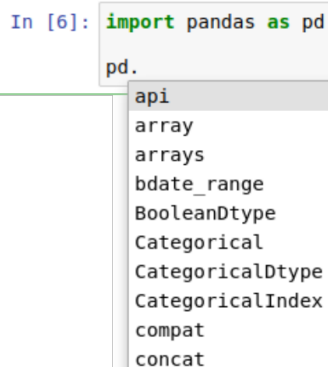
... for objects with methods and attributes:

```
In [ ]: data = []
data.
```



... and also for modules:

```
In [6]: import pandas as pd
pd.
```



Note:

As you may have noticed in surprise, the `__version__` method used above is not offered in the selection. IPython initially hides these private methods and attributes that begin with underscores. However, they can also be completed with a tabulator if you first enter an underscore. Alternatively, you can change this setting in the IPython configuration.

... for almost everything:

```
In [ ]: path = './'
```

- ./config.rst
- ./examples.ipynb
- ./index.rst
- ./shortcuts.rst
- ./tab-completion-for-modules.png
- ./tab-completion-for-objects.png

Displaying information about an object

With a question mark (?) you can display information about an object if, for example, there is a method `multiply` with the following docstring:

```
[8]: import numpy as np
```

```
[9]: np.mean?
```

```
Signature:
np.mean(
    a,
    axis=None,
    dtype=None,
    out=None,
    keepdims=<no value>,
    *,
    where=<no value>,
)
Docstring:
Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over
the flattened array by default, otherwise over the specified axis.
`float64` intermediate and return values are used for integer inputs.

Parameters
-----
a : array_like
    Array containing numbers whose mean is desired. If `a` is not an
    array, a conversion is attempted.
axis : None or int or tuple of ints, optional
    Axis or axes along which the means are computed. The default is to
    compute the mean of the flattened array.

.. versionadded:: 1.7.0
```

(continues on next page)

(continued from previous page)

If this is a tuple of ints, a mean is performed over multiple axes, instead of a single axis or all the axes as before.

`dtype` : data-type, optional

Type to use in computing the mean. For integer inputs, the default is `'float64'`; for floating point inputs, it is the same as the input dtype.

`out` : ndarray, optional

Alternate output array in which to place the result. The default is `'None'`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See [:ref:'ufuncs-output-type'](#) for more details.

`keepdims` : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then `'keepdims'` will not be passed through to the `'mean'` method of sub-classes of `'ndarray'`, however any non-default value will be. If the sub-class' method does not implement `'keepdims'` any exceptions will be raised.

`where` : array_like of bool, optional

Elements to include in the mean. See `~numpy.ufunc.reduce` for details.

.. **versionadded::** 1.20.0

Returns

`m` : ndarray, see dtype parameter above

If `'out=None'`, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See Also

`average` : Weighted average

`std`, `var`, `nanmean`, `nanstd`, `nanvar`

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `'float32'` (see example below). Specifying a higher-precision accumulator using the `'dtype'` keyword can alleviate this issue.

By default, `'float16'` results are computed using `'float32'` intermediates for extra precision.

(continues on next page)

(continued from previous page)

Examples

```

>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([2., 3.])
>>> np.mean(a, axis=1)
array([1.5, 3.5])

```

In single precision, `mean` can be inaccurate:

```

>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.54999924

```

Computing the mean in float64 is more accurate:

```

>>> np.mean(a, dtype=np.float64)
0.550000000074505806 # may vary

```

Specifying a `where` argument:

```

>>> a = np.array([[5, 9, 13], [14, 10, 12], [11, 15, 19]])
>>> np.mean(a)
12.0
>>> np.mean(a, where=[[True], [False], [False]])
9.0

```

```

File:      ~/spack/var/spack/environments/python-38/.spack-env/view/lib/python3.8/site-
packages/numpy/core/fromnumeric.py
Type:      function

```

By using `??` the source code of the function is also displayed, if this is possible:

```
[10]: np.mean??
```

Signature:

```

np.mean(
    a,
    axis=None,
    dtype=None,
    out=None,
    keepdims=<no value>,
    *,
    where=<no value>,
)

```

Source:

```

@array_function_dispatch(_mean_dispatcher)
def mean(a, axis=None, dtype=None, out=None, keepdims=np._NoValue, *,
        where=np._NoValue):

```

(continues on next page)

(continued from previous page)

```

"""
Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over
the flattened array by default, otherwise over the specified axis.
`float64` intermediate and return values are used for integer inputs.

Parameters
-----
a : array_like
    Array containing numbers whose mean is desired. If `a` is not an
    array, a conversion is attempted.
axis : None or int or tuple of ints, optional
    Axis or axes along which the means are computed. The default is to
    compute the mean of the flattened array.

    .. versionadded:: 1.7.0

    If this is a tuple of ints, a mean is performed over multiple axes,
    instead of a single axis or all the axes as before.
dtype : data-type, optional
    Type to use in computing the mean. For integer inputs, the default
    is `float64`; for floating point inputs, it is the same as the
    input dtype.
out : ndarray, optional
    Alternate output array in which to place the result. The default
    is ``None``; if provided, it must have the same shape as the
    expected output, but the type will be cast if necessary.
    See :ref:`ufuncs-output-type` for more details.

keepdims : bool, optional
    If this is set to True, the axes which are reduced are left
    in the result as dimensions with size one. With this option,
    the result will broadcast correctly against the input array.

    If the default value is passed, then `keepdims` will not be
    passed through to the `mean` method of sub-classes of
    `ndarray`, however any non-default value will be. If the
    sub-class' method does not implement `keepdims` any
    exceptions will be raised.

where : array_like of bool, optional
    Elements to include in the mean. See `~numpy.ufunc.reduce` for details.

    .. versionadded:: 1.20.0

Returns
-----
m : ndarray, see dtype parameter above
    If `out=None`, returns a new array containing the mean values,
    otherwise a reference to the output array is returned.

```

(continues on next page)

(continued from previous page)

See Also

average : Weighted average
 std, var, nanmean, nanstd, nanvar

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `'float32'` (see example below). Specifying a higher-precision accumulator using the `'dtype'` keyword can alleviate this issue.

By default, `'float16'` results are computed using `'float32'` intermediates for extra precision.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([2., 3.])
>>> np.mean(a, axis=1)
array([1.5, 3.5])
```

In single precision, `'mean'` can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.549999924
```

Computing the mean in float64 is more accurate:

```
>>> np.mean(a, dtype=np.float64)
0.550000000074505806 # may vary
```

Specifying a `where` argument:

```
>>> a = np.array([[5, 9, 13], [14, 10, 12], [11, 15, 19]])
>>> np.mean(a)
12.0
>>> np.mean(a, where=[[True], [False], [False]])
9.0
```

```
"""
```

```
kwargs = {}
if keepdims is not np._NoValue:
```

(continues on next page)

(continued from previous page)

```

        kwargs['keepdims'] = keepdims
    if where is not np._NoValue:
        kwargs['where'] = where
    if type(a) is not mu.ndarray:
        try:
            mean = a.mean
        except AttributeError:
            pass
        else:
            return mean(axis=axis, dtype=dtype, out=out, **kwargs)

    return _methods._mean(a, axis=axis, dtype=dtype,
                          out=out, **kwargs)
File:      ~/spack/var/spack/environments/python-38/.spack-env/view/lib/python3.8/site-
↳ packages/numpy/core/fromnumeric.py
Type:      function

```

? can also be used to search in the IPython namespace. In doing so, a series of characters can be represented with the wildcard (*). For example, to get a list of all functions in the top-level NumPy namespace that contain mean:

```
[11]: np.*mean*?
```

```

np.mean
np.nanmean

```

2.1.3 IPython magic

IPython not only enables Python to be used interactively, but also extends the Python syntax with so-called *magic commands*, which are provided with the prefix %. They are designed to quickly and easily solve common data analysis problems. A distinction is made between two different types of *magic commands*:

- *line magics*, denoted by a single % prefix, that run on a single input line
- *cell magics* which are preceded by a double symbol %% and which are executed within a notebook cell.

Execute external code: %run

If you start developing larger code, you will likely be working in both IPython for interactive exploration and a text editor to save code that you want to reuse. With the %run magic you can execute this code directly in your IPython session.

Imagine you created a `myscript.py` file with the following content:

```

def square(x):
    return x**2

for N in range(1, 4):
    print(N, "squared is", square(N))

```

```
[1]: %run myscript.py
```

```
1 squared is 1
2 squared is 4
3 squared is 9
```

Note that after running this script, all of the functions defined in it will be available for use in your IPython session:

```
[2]: square(4)
```

```
[2]: 16
```

There are several ways you can improve the way your code runs. As usual, you can display the documentation in IPython with `%run?`.

Run timing code: `%timeit`

Another example of a Magic function is `%timeit`, which automatically determines the execution time of the following one-line Python statement. So we can e.g. output the performance of a list comprehension with:

```
[3]: %timeit L = [n ** 2 for n in range(1000)]
```

```
27.6 µs ± 290 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

The advantage of `%timeit` is that short commands automatically run multiple runs to get more robust results. For multi-line instructions, adding a second `%` character creates cell magic that can process multiple input lines. For example, here is the equivalent construction using a `for` loop:

```
[4]: %%timeit
L = []
for n in range(1000):
    L.append(n ** 2)
```

```
29.7 µs ± 207 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

We can immediately see that the list comprehension is about 10% faster than its equivalent with a `for` loop. We then describe performance measurements and optimisations in more detail in [Profiling](#).

Execute code from other interpreters

IPython has a `%%script` script magic with which you can execute a cell in a subprocess of an interpreter on your system, e.g. `bash`, `ruby`, `perl`, `zsh`, `R` etc. This can also be its own script that expects input in `stdin`. To do this, simply pass a path or a shell command to the program that is specified in the `%%script` line. The rest of the cell is executed by this script, capturing `stdout` or `err` from the subprocess and displaying it.

```
[5]: %%script python2
import sys
```

```
print("Python: %s" % sys.version)
```

```
Python 2.7.15 (default, Oct 22 2018, 19:33:46)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)]
```

```
[6]: %%script python3
import sys

print("Python: %s" % sys.version)

Python: 3.11.4 (main, Jun 15 2023, 07:55:38) [Clang 14.0.3 (clang-1403.0.22.14.1)]
```

```
[7]: %%ruby
puts "Ruby #{RUBY_VERSION}"

Ruby 2.6.10
```

```
[8]: %%bash
echo "$BASH"

/bin/bash
```

You can capture `stdout` and `err` from these sub-processes in Python variables:

```
[9]: %%bash --out output --err error
echo "stdout"
echo "stderr" >&2
```

```
[10]: print(error)
print(output)

stderr

stdout
```

Configure standard script magic

The list of aliases for the `script` magic is configurable. By default, some common interpreters can be used if necessary. However, you can also specify your own interpreter in `ipython_config.py`:

```
c.ScriptMagics.scripts = ["R", "pypy", "myprogram"]
c.ScriptMagics.script_paths = {"myprogram": "/path/to/myprogram"}
```

Help functions: `?`, `%magic` and `%lsmagic`

Like normal Python functions, the IPython magic functions have docstrings that can be easily accessed. E.g. to read the documentation of the `%timeit` magic, just type:

```
[11]: %timeit?
```

Documentation for other functions can be accessed in a similar manner. To access a general description of the `%magic` functions available, including some examples, you can type:

```
[12]: %magic
```

For a quick list of all available magic functions, type:

```
[13]: %lsmagic
```

```
[13]: Available line magics:
```

```
%alias %alias_magic %autoawait %autocall %automagic %autosave %bookmark %cat %cd_
→ %clear %colors %conda %config %connect_info %cp %debug %dhist %dirs %doctest_
→ mode %ed %edit %env %gui %hist %history %killbgscripts %ldir %less %lf %lk
→ %ll %load %load_ext %loadpy %logoff %logon %logstart %logstate %logstop %ls
→ %lsmagic %lx %macro %magic %man %matplotlib %mkdir %more %mv %notebook %page_
→ %pastebin %pdb %pdef %pdoc %pfile %pinfo %pinfo2 %pip %popd %pprint
→ %precision %prun %psearch %psource %pushd %pwd %pycat %pylab %qtconsole
→ %quickref %recall %rehashx %reload_ext %rep %rerun %reset %reset_selective %rm_
→ %rmdir %run %save %sc %set_env %store %sx %system %tb %time %timeit
→ %unalias %unload_ext %who %who_ls %whos %xdel %xmode
```

```
Available cell magics:
```

```
%%! %%HTML %%SVG %%bash %%capture %%debug %%file %%html %%javascript %%js %
→ %latex %%markdown %%perl %%prun %%pypy %%python %%python2 %%python3 %%ruby %
→ %script %%sh %%svg %%sx %%system %%time %%timeit %%writefile
```

```
Automagic is ON, % prefix IS NOT needed for line magics.
```

You can also simply define your own magic functions. For more information, see [Defining custom magics](#).

2.1.4 Shell commands in IPython

The IPython Notebook allows simple UNIX/Linux commands to be executed in a single input cell. There are no limits but when using, please keep in mind that in contrast to a regular UNIX/Linux shell, start each shell command with a `!`, for example `!ls` for the command `ls` (see below for further explanations about the command). Furthermore, each shell command is executed in its own subshell. For this reason, the results of previous shell commands are not available to you.

To begin with, the command `ls` lists the files in the current working directory. The output is shown below the input cell, and lists the single file `shell.ipynb`:

```
[1]: !ls
```

```
debugging.ipynb      myscript.py
display.ipynb        shell.ipynb
examples.ipynb       start.rst
extensions.rst       tab-completion-for-anything.png
importing.ipynb      tab-completion-for-modules.png
index.rst            tab-completion-for-objects.png
magics.ipynb         unix-shell
mypackage
```

The command `!pwd` displays the path to working directory:

```
[2]: !pwd
```

```
/Users/veit/cusy/trn/Python4DataScience/docs/workspace/ipython
```

The command `!echo` outputs text given as parameter to the `echo` command. The example below demonstrates how to print `Hello world`:

```
[3]: !echo "Hello world!"
```

```
Hello world!
```

Passing values to and from the shell

There is a clever way through which you can access the output of a UNIX/Linux command as a variable in Python. Assign the output of a UNIX/Linux command to a variable as follows:

```
[4]: contents = !ls
```

Here the Python variable `contents` has been assigned the output of the command `ls`. As a result, `contents` is a list, where each list element corresponds to a line in the output. With the `print` command you output the list contents:

```
[5]: print(contents)
```

```
['debugging.ipynb', 'display.ipynb', 'examples.ipynb', 'extensions.rst', 'importing.ipynb',  
↪ 'index.rst', 'magics.ipynb', '\\x1b[34mmypackage\\x1b[m\\x1b[m', 'myscript.py', 'shell.  
↪ ipynb', 'start.rst', '\\x1b[31mtab-completion-for-anything.png\\x1b[m\\x1b[m', '\\  
↪ x1b[31mtab-completion-for-modules.png\\x1b[m\\x1b[m', '\\x1b[31mtab-completion-for-  
↪ objects.png\\x1b[m\\x1b[m', '\\x1b[34munix-shell\\x1b[m\\x1b[m']]
```

You will see the same result below when executing the `pwd` command. The current directory is stored in the variable `directory`:

```
[6]: directory = !pwd
```

```
[7]: print(directory)
```

```
['/Users/veit/cusy/trn/Python4DataScience/docs/workspace/ipython']
```

2.1.5 Unix shell

Any command on the command line will also work in Jupyter Notebooks if prefixed with `!`. The results can then interact with the Jupyter namespace, see [Passing values to and from the shell](#).

Navigate through files and directories

First let us find out where we are by running a command called `pwd`:

```
[1]: !pwd
```

```
/Users/veit/cusy/trn/Python4DataScience/docs/workspace/ipython/unix-shell
```

Here, the response is the iPython chapter of the Jupyter tutorial in my home directory `/Users/veit`.

On Windows the home directory will look like `C:\Documents and Settings\veit` or `C:\Users\veit` and on Linux like `/home/veit`.

To see the contents of our directory, we can use `ls`:

```
[2]: !ls
```

create-delete.ipynb	grep-find.ipynb	pipes-filters.ipynb
file-system.ipynb	index.rst	shell-variables.ipynb

- a trailing / indicates a directory
- @ indicates a link
- * indicates an executable

Depending on your default options, the shell might also use colors to indicate whether an entry is a file or a directory.

ls options and arguments

```
[3]: !ls -F ../
debugging.ipynb          myscript.py
display.ipynb            shell.ipynb
examples.ipynb           start.rst
extensions.rst            tab-completion-for-anything.png*
importing.ipynb          tab-completion-for-modules.png*
index.rst                 tab-completion-for-objects.png*
magics.ipynb             unix-shell/
mypackage/
```

ls is the command, with the option -F and the argument ../.

- Options either start with a single dash (-) or two dashes (--), and they change the behavior of a command.
- Arguments tell the command what to operate on.
- Options and arguments are sometimes also referred as parameters.
- Each part is separated by spaces.
- Also, capitalisation is important, for example
 - ls -s will display the size of files and directories alongside the names,
 - while ls -S will sort the files and directories by size.

```
[4]: !ls -s
total 184
24 create-delete.ipynb   24 grep-find.ipynb      16 pipes-filters.ipynb
96 file-system.ipynb     8 index.rst             16 shell-variables.ipynb
```

```
[5]: !ls -S
file-system.ipynb   create-delete.ipynb  shell-variables.ipynb
grep-find.ipynb    pipes-filters.ipynb  index.rst
```

Show all options and arguments

ls comes with a lot of other useful options. Using man you can print out the built-in manual page for the desired UNIX/Linux-command:

[6]: !man ls

```
LS(1)                                General Commands Manual                                LS(1)

NAME
    ls - list directory contents

SYNOPSIS
    ls [-@ABCFGHILOPRSTUWabcdefghiklmnopqrstuvwxy1%,] [--color=____]
        [-D ____] [____ ____]

DESCRIPTION
    For each operand that names a ____ of a type other than directory, ls
    displays its name as well as any requested, associated information. For
    each operand that names a ____ of type directory, ls displays the names
    of files contained within that directory, as well as any requested,
    associated information.

    If no operands are given, the contents of the current directory are
    displayed. If more than one operand is given, non-directory operands are
    displayed first; directory and non-directory operands are sorted
    separately and in lexicographical order.

    The following options are available:

    -@      Display extended attribute keys and sizes in long (-l) output.

    ...

macOS 13.4                        August 31, 2020                        macOS 13.4
```

Illegal options

If you try to use an option that isn't supported, the commands will usually print an error message, for example for:

[7]: !ls -z

```
ls: invalid option -- z
usage: ls [-@ABCFGHILOPRSTUWabcdefghiklmnopqrstuvwxy1%,] [--color=when] [-D format]_
↪ [file ...]
```


Hidden Files

With the `-a` option you can display all files:

```
[8]: !ls -a
.          create-delete.ipynb  index.rst
..         file-system.ipynb   pipes-filters.ipynb
.ipynb_checkpoints  grep-find.ipynb  shell-variables.ipynb
```

In addition to the hidden directories `..` and `.`, you may also see a directory called `.ipynb_checkpoints`. This file usually contains snapshots of the Jupyter notebooks.

Show directory tree The command `tree` lists contents of directories in a tree-like format.

```
[9]: !tree
.
├── create-delete.ipynb
├── file-system.ipynb
├── grep-find.ipynb
├── index.rst
├── pipes-filters.ipynb
└── shell-variables.ipynb

1 directory, 6 files
```

Change directory

At first it may seem irritating to some that they cannot use `!cd` to change to another directory.

```
[10]: !pwd
/Users/veit/cusy/trn/Python4DataScience/docs/workspace/ipython/unix-shell
```

```
[11]: !cd ..
```

```
[12]: !pwd
/Users/veit/cusy/trn/Python4DataScience/docs/workspace/ipython/unix-shell
```

The reason for this is that Jupyter uses a temporary subshell. If you want to change to another directory permanently, you have to use the *magic command* `%cd`.

```
[13]: %cd ..
/Users/veit/cusy/trn/Python4DataScience/docs/workspace/ipython
```

```
[14]: !pwd
/Users/veit/cusy/trn/Python4DataScience/docs/workspace/ipython
```

With the `%automagic` function, these can also be used without the preceding `%` character:

```
[16]: %automagic

Automagic is ON, % prefix IS NOT needed for line magics.
```

```
[17]: cd ..

/Users/veit/cusy/trn/Python4DataScience/docs/workspace
```

Absolute and relative Paths

```
[18]: cd .

/Users/veit/cusy/trn/Python4DataScience/docs/workspace
```

```
[19]: cd ../../..

/Users/veit/cusy/trn/Python4DataScience
```

```
[20]: cd ..

/Users/veit/cusy/trn
```

```
[21]: cd /

/
```

```
[22]: cd

/Users/veit
```

```
[23]: cd ~

/Users/veit
```

```
[24]: cd /Users/veit

/Users/veit
```

Create, update and delete files and directories

Creates a new directory `test` and then checks this with `ls`:

```
[1]: !mkdir tests
```

```
[2]: !ls

create-delete.ipynb  index.rst          tests
file-system.ipynb   pipes-filters.ipynb
grep-find.ipynb     shell-variables.ipynb
```

Then we create the file `test_file.txt` in this directory.

```
[3]: !touch tests/test_file.txt
```

```
[4]: !ls tests
test_file.txt
```

Now we change the suffix of the file:

```
[5]: !mv tests/test_file.txt tests/test_file.py
```

```
[6]: !ls tests
test_file.py
```

Now we make a copy of this file:

```
[7]: !cp tests/test_file.py tests/test_file2.py
```

```
[8]: !ls tests
test_file.py  test_file2.py
```

A directory with all the files it contains is also possible recursively with the `-r` option:

```
[9]: !cp -r tests tests.bak
```

```
[10]: !ls tests.bak
test_file.py  test_file2.py
```

Finally, we delete the directories `tests` and `tests.bak` again:

```
[11]: !rm -r tests tests.bak
```

```
[12]: !ls
create-delete.ipynb  grep-find.ipynb  pipes-filters.ipynb
file-system.ipynb   index.rst        shell-variables.ipynb
```

Transferring files

wget

```
[13]: !wget https://dvc.org/deb/dvc.list
--2023-07-19 17:00:21-- https://dvc.org/deb/dvc.list
Auflösen des Hostnamens dvc.org (dvc.org)... 2606:4700:3036::6815:51cd, 2606:4700:3033::
ac43:a44c, 172.67.164.76, ...
Verbindungsaufbau zu dvc.org (dvc.org)|2606:4700:3036::6815:51cd|:443 ... verbunden.
HTTP-Anforderung gesendet, auf Antwort wird gewartet ... 303 See Other
Platz: https://s3-us-east-2.amazonaws.com/dvc-s3-repo/deb/dvc.list [folgend]
--2023-07-19 17:00:21-- https://s3-us-east-2.amazonaws.com/dvc-s3-repo/deb/dvc.list
Auflösen des Hostnamens s3-us-east-2.amazonaws.com (s3-us-east-2.amazonaws.com)... 52.
```

(continues on next page)

(continued from previous page)

```

→ 219.100.82
Verbindungsaufbau zu s3-us-east-2.amazonaws.com (s3-us-east-2.amazonaws.com) | 52.219.100.
→ 82|:443 ... verbunden.
HTTP-Anforderung gesendet, auf Antwort wird gewartet ... 200 OK
Länge: 51 [binary/octet-stream]
Wird in »dvc.list« gespeichert.

dvc.list          100%[=====>]          51  --.-KB/s    in 0s

2023-07-19 17:00:22 (1,13 MB/s) - »dvc.list« gespeichert [51/51]

```

- `-r` recursively crawls other files and directories
- `-np` avoids crawling to parent directories
- `-D` targets only the following domain name
- `-nH` avoids creating a subdirectory for the websites content
- `-m` mirrors with time stamping, time stamping, infinite recursion depth, and preservation of FTP directory settings
- `-q` suppresses the output to the screen

cURL

Alternatively, you can use cURL, which supports a much larger range of protocols.

```
[14]: !curl -o dvc.list https://dvc.org/deb/dvc.list
```

% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Current
			Dload Upload	Total	Spent	Left	Speed
100	85	100	85	0	0	251	0
--:--:--	--:--:--	--:--:--	--:--:--	--:--:--	--:--:--	--:--:--	254

Pipes and filters

`ls` shows all files and directories at this point.

```
[1]: !ls
```

create-delete.ipynb	grep-find.ipynb	shell-variables.ipynb
dvc.list	index.rst	
file-system.ipynb	pipes-filters.ipynb	

With `*.rst` we restrict the results to all files with the suffix `.rst`:

```
[2]: !ls *.rst
```

```
index.rst
```

We can also output only the number of lines, words and characters in these documents:

```
[3]: !wc *.rst
```

18	48	450	index.rst
----	----	-----	-----------

Now we write the number of characters in the file `length.txt` and then output the text with `cat`:

```
[4]: !wc -m *.rst > length.txt
```

```
[5]: !cat length.txt
```

```
450 index.rst
```

We can also have the files sorted by the number of characters:

```
[6]: !sort -n length.txt
```

```
450 index.rst
```

```
[7]: !sort -n length.txt > sorted-length.txt
```

We can also overwrite the existing file:

```
[8]: !sort -n length.txt > length.txt
```

If we only want to know the total number of characters, i.e. only output the last line, we can do this with `tail`:

```
[9]: !tail -n 1 length.txt
```

`>` is used to overwrite a file while `>>` is used to append to a file.

```
[10]: !echo amount of characters >> length.txt
```

```
[11]: !cat length.txt
```

```
amount of characters
```

Pipe |

You can connect commands with a pipe (`|`). In the following one-liner, we want to display the number of characters for the shortest file:

```
[12]: !wc -l *.rst | sort -n | head
```

```
18 index.rst
```

If we want to display the first lines of the main text (without the first three lines for the title):

```
[13]: !cat index.rst | head -n 5 | tail -n 2
```

Any command on the command line will also work in Jupyter Notebooks if prefixed with ```!```. The results can then interact with the Jupyter namespace, see

grep and find

grep

grep finds and prints lines in files that match a [regular expression](#). In the following example, we search for the string Python:

```
[1]: !grep Python ../index.rst
```

```
IPython
`IPython <https://ipython.org/>`, or *Interactive Python*, was initially an
advanced Python interpreter that has now grown into an extensive project
Today, IPython is not only an interactive interface to Python, but also offers a
number of useful syntactic additions for the language. In addition, IPython is
    * `Miki Tebeka - IPython: The Productivity Booster
```

The option `-w` limits the matches to the word boundaries so that IPython is ignored:

```
[2]: !grep -w Python ../index.rst
```

```
`IPython <https://ipython.org/>`, or *Interactive Python*, was initially an
advanced Python interpreter that has now grown into an extensive project
Today, IPython is not only an interactive interface to Python, but also offers a
```

`-n` shows the line numbers that match:

```
[3]: !grep -n -w Python ../index.rst
```

```
4:`IPython <https://ipython.org/>`, or *Interactive Python*, was initially an
5:advanced Python interpreter that has now grown into an extensive project
7:Today, IPython is not only an interactive interface to Python, but also offers a
```

`-v` inverts our search

```
[4]: !grep -n -v "^ " ../index.rst
```

```
1:IPython
2:=====
3:
4:`IPython <https://ipython.org/>`, or *Interactive Python*, was initially an
5:advanced Python interpreter that has now grown into an extensive project
6:designed to provide tools for the entire life cycle of research computing.
7:Today, IPython is not only an interactive interface to Python, but also offers a
8:number of useful syntactic additions for the language. In addition, IPython is
9:closely related to the `Jupyter project <https://jupyter.org/>`.
10:
11:... seealso::
14:
15:... toctree::
19:
```

grep has lots of other options. To find out what they are, you can type:

```
[5]: !grep --help
```

```
usage: grep [-abcdDEFGHhIiJlLMmnOopqRSsUVvwXxZz] [-A num] [-B num] [-C[num]]
          [-e pattern] [-f file] [--binary-files=value] [--color=when]
          [--context[=num]] [--directories=action] [--label] [--line-buffered]
          [--null] [pattern] [file ...]
```

In the following example we use the `-E` option and put the pattern in quotes to prevent the shell from trying to interpret it. The `^` in the pattern anchors the match to the start of the line and the `.` matches a single character.

```
[6]: !grep -n -E "^Python" ../index.rst
1:IPython
```

find

`find .` searches in this directory whereby the search is restricted to directories with `-type d`.

```
[7]: !find .. -type d
..
../mypackage
../unix-shell
../unix-shell/.ipynb_checkpoints
../.ipynb_checkpoints
```

With `-type f` the search is restricted to files.

```
[8]: !find . -type f
./index.rst
./sorted-length.txt
./create-delete.ipynb
./length.txt
./dvc.list
./file-system.ipynb
./pipes-filters.ipynb
./shell-variables.ipynb
./.ipynb_checkpoints/create-delete-checkpoint.ipynb
./.ipynb_checkpoints/grep-find-checkpoint.ipynb
./.ipynb_checkpoints/pipes-filters-checkpoint.ipynb
./.ipynb_checkpoints/file-system-checkpoint.ipynb
./grep-find.ipynb
```

With `-mtime` the search is limited to the last `X` days, in our example to the last day:

```
[9]: !find . -mtime -1
.
./sorted-length.txt
./create-delete.ipynb
./length.txt
./dvc.list
./file-system.ipynb
./pipes-filters.ipynb
./.ipynb_checkpoints
```

(continues on next page)

(continued from previous page)

```
./ipynb_checkpoints/create-delete-checkpoint.ipynb
./ipynb_checkpoints/grep-find-checkpoint.ipynb
./ipynb_checkpoints/pipes-filters-checkpoint.ipynb
./ipynb_checkpoints/file-system-checkpoint.ipynb
./grep-find.ipynb
```

With `-name` you can filter the search by name.

```
[10]: !find .. -name "*.rst"

../index.rst
../unix-shell/index.rst
../extensions.rst
../start.rst
```

Now we count the characters in the files with the suffix `.rst`:

```
[11]: !wc -c $(find .. -name "*.rst")

 833 ../index.rst
 450 ../unix-shell/index.rst
2097 ../extensions.rst
1145 ../start.rst
4525 total
```

It is also possible to search for a regular expression in these files:

```
[12]: !grep "ipython.org" $(find .. -name "*.rst")

../index.rst: IPython <https://ipython.org/>`, or *Interactive Python*, was initially an
```

Finally, we filter out all results whose path contains `ipynb_checkpoints`:

```
[13]: !find . -name "*.ipynb" | grep -v ipynb_checkpoints

./create-delete.ipynb
./file-system.ipynb
./pipes-filters.ipynb
./shell-variables.ipynb
./grep-find.ipynb
```

Shell variables

Display of all shell variables

```
[1]: !set

...
HOME=/Users/veit
...
PATH=/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/bin:/opt/homebrew/Cellar/
↳ pipenv/2023.6.18/libexec/tools:/Users/veit/spack/bin:/opt/homebrew/bin:/opt/homebrew/
↳ sbin:/usr/local/bin:/System/Cryptexes/App/usr/bin:/usr/bin:/bin:/usr/sbin:/sbin:/
```

(continues on next page)

(continued from previous page)

```

↪Library/TeX/texbin:/usr/local/MacGPG2/bin:/Library/Apple/usr/bin:/var/run/com.apple.
↪security.cryptexd/codex.system/bootstrap/usr/local/bin:/var/run/com.apple.security.
↪cryptexd/codex.system/bootstrap/usr/bin:/var/run/com.apple.security.cryptexd/codex.
↪system/bootstrap/usr/appleinternal/bin
...

```

Showing the value of a variable

```

[2]: !echo $HOME
/Users/veit

```

The path variable

It defines the shell's search path, i.e., the list of directories that the shell looks in for runnable programs.

```

[3]: !echo $PATH
/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/bin:/opt/homebrew/Cellar/pipenv/
↪2023.6.18/libexec/tools:/Users/veit/spack/bin:/opt/homebrew/bin:/opt/homebrew/sbin:/
↪usr/local/bin:/System/Cryptexes/App/usr/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/
↪texbin:/usr/local/MacGPG2/bin:/Library/Apple/usr/bin:/var/run/com.apple.security.
↪cryptexd/codex.system/bootstrap/usr/local/bin:/var/run/com.apple.security.cryptexd/
↪codex.system/bootstrap/usr/bin:/var/run/com.apple.security.cryptexd/codex.system/
↪bootstrap/usr/appleinternal/bin

```

Creating and changing variables

Creating or overwriting variables

```

[4]: !export SPACK_ROOT=~/.spack

```

Append additional specifications

```

[5]: !export PATH=/usr/local/opt/python@3.7/bin:$PATH

```

2.1.6 Show objects with display

IPython can display objects such as HTML, JSON, PNG, JPEG, SVG and Latex

Images

To display images (JPEG, PNG) in IPython and notebooks, you can use the `Image` class:

```
[1]: from IPython.display import Image
Image('https://www.python.org/images/python-logo.gif')
```

```
[1]: <IPython.core.display.Image object>
```

```
[2]: from IPython.display import SVG
SVG('https://upload.wikimedia.org/wikipedia/commons/c/c3/Python-logo-notext.svg')
```

```
[2]:
```

Non-embedded images

- By default, image data is embedded:

```
Image ('img_url')
```

- However, if the url is given as kwarg, this is interpreted as a soft link:

```
Image (url='img_url')
```

- `embed` can also be specified explicitly:

```
Image (url='img_url', embed = True)
```

HTML

Python objects can declare HTML representations to be displayed in a notebook:

```
[3]: from IPython.display import HTML
```

```
[4]: %%html
<ul>
  <li>foo</li>
  <li>bar</li>
</ul>
```

```
<IPython.core.display.HTML object>
```

Javascript

With notebooks, objects can also declare a JavaScript representation. This enables for example data visualisations with Javascript libraries like `d3.js`.

```
[5]: from IPython.display import Javascript
```

```
welcome = Javascript(
    'alert("Dies ist ein Beispiel für eine durch IPython angezeigte Javascript-Warnung.")
```

(continues on next page)

(continued from previous page)

```
↩ '
)
display(welcome)

<IPython.core.display.Javascript object>
```

For more extensive Javascript you can also use the `%%javascript` syntax.

LaTeX

`IPython.display` also has built-in support for displaying mathematical expressions set in LaTeX and rendered in the browser with [MathJax](#):

```
[6]: from IPython.display import Math

Math(r"F(k) = \int_{-\infty}^{\infty} f(x) e^{2\pi i k} dx")
```

```
[6]: 
$$F(k) = \int_{-\infty}^{\infty} f(x) e^{2\pi i k} dx$$

```

With the `Latex` class you have to specify the limits yourself. In this way, however, you can also use other LaTeX modes, such as `eqnarray`:

```
[7]: from IPython.display import Latex

Latex(
    r"""\begin{eqnarray}
\nabla \times \vec{\mathbf{B}} = -\frac{1}{c} \frac{\partial \vec{\mathbf{E}}}{\partial t}, \quad \frac{\partial \vec{\mathbf{E}}}{\partial t} = \frac{1}{c} \nabla \times \vec{\mathbf{B}} \\
\frac{\partial \vec{\mathbf{B}}}{\partial t} = -\frac{1}{c} \nabla \times \vec{\mathbf{E}} \\
\end{eqnarray}""")
```

```
[7]: 
$$\nabla \times \vec{\mathbf{B}} - \frac{1}{c} \frac{\partial \vec{\mathbf{E}}}{\partial t} = \frac{4\pi}{c} \vec{\mathbf{j}} \quad (2.1)$$


$$\frac{\partial \vec{\mathbf{B}}}{\partial t} = -\frac{1}{c} \nabla \times \vec{\mathbf{E}} \quad (2.2)$$

```

Audio

IPython also enables interactive work with sounds. With the `display.Audio` class you can create an audio control that is embedded in the notebook. The interface is analogous to that of the `Image` class. All audio formats supported by the browser can be used.

```
[8]: from IPython.display import Audio
```

In the following we will output the sine function of a NumPy array as an audio signal. The `Audio` class normalises and codes the data and embeds the resulting audio in the notebook.

```
[9]: import numpy as np

f = 500.0
rate = 8000
L = 3
times = np.linspace(0, L, rate * L)
signal = np.sin(f * times)

Audio(data=signal, rate=rate)

[9]: <IPython.lib.display.Audio object>
```

Links to local files

IPython has built-in classes for generating links to local files. To do this, create a link to a single file with the `FileLink` object:

```
[10]: from IPython.display import FileLink, FileLinks

FileLink("magics.ipynb")

[10]: /Users/veit/cusy/trn/Python4DataScience/docs/workspace/ipython/magics.ipynb
```

Alternatively, you can generate a list with links to all files in a directory, e.g.:

```
[11]: FileLinks(".")

[11]: ./
      index.rst
      tab-completion-for-modules.png
      tab-completion-for-objects.png
      tab-completion-for-anything.png
      debugging.ipynb
      magics.ipynb
      shell.ipynb
      display.ipynb
      examples.ipynb
      myscript.py
      importing.ipynb
      extensions.rst
      start.rst
./ipynb_checkpoints/
  display-checkpoint.ipynb
./mypackage/
  __init__.py
  foo.ipynb
./unix-shell/
  index.rst
  create-delete.ipynb
  file-system.ipynb
  pipes-filters.ipynb
```

(continues on next page)

(continued from previous page)

```
shell-variables.ipynb
grep-find.ipynb
```

Display notebooks

```
[12]: import os
import sys
import types

import nbformat

from IPython.display import HTML, display
from pygments import highlight
from pygments.formatters import HtmlFormatter
from pygments.lexers import PythonLexer

formatter = HtmlFormatter()
lexer = PythonLexer()

# publish the CSS for pygments highlighting
display(
    HTML(
        """
<style type='text/css'>
%s
</style>
"""
        % formatter.get_style_defs()
    )
)

<IPython.core.display.HTML object>
```

```
[13]: def show_notebook(fname):
    """display a short summary of the cells of a notebook"""
    nb = nbformat.read(fname, as_version=4)
    html = []
    for cell in nb.cells:
        html.append("<h4>%s cell</h4>" % cell.cell_type)
        if cell.cell_type == "code":
            html.append(highlight(cell.source, lexer, formatter))
        else:
            html.append("<pre>%s</pre>" % cell.source)
    display(HTML("\n".join(html)))

show_notebook(os.path.join("mypackage/foo.ipynb"))

<IPython.core.display.HTML object>
```

2.1.7 foo.ipynb

```
[1]: def bar():  
     return "bar"
```

```
[2]: def dirlist():  
     listing = !ls  
     return listing
```

```
[3]: def whatsmyname():  
     return __name__
```

2.1.8 Import notebooks

To be able to develop more modularly, the import of notebooks is necessary. However, since notebooks are not Python files, they are not easy to import. Fortunately, Python provides some hooks for the import so that IPython notebooks can eventually be imported.

```
[1]: import os  
     import sys  
     import types
```

```
[2]: import nbformat  
  
from IPython import get_ipython  
from IPython.core.interactiveshell import InteractiveShell
```

Import hooks usually have two objects:

- **Module Loader** that takes a module name (e.g. `IPython.display`) and returns a module
- **Module Finder**, which finds out if a module is present and tells Python which *loader* to use

But first, let's write a method that a notebook will find using the fully qualified name and the optional path. E.g. `mypackage.foo` becomes `mypackage/foo.ipynb` and replaces `Foo_Bar` with `Foo Bar` if `Foo_Bar` doesn't exist.

```
[3]: def find_notebook(fullname, path=None):  
     name = fullname.rsplit(".", 1)[-1]  
     if not path:  
         path = []  
     for d in path:  
         nb_path = os.path.join(d, name + ".ipynb")  
         if os.path.isfile(nb_path):  
             return nb_path  
     # let import Foo_Bar find "Foo Bar.ipynb"  
     nb_path = nb_path.replace("_", " ")  
     if os.path.isfile(nb_path):  
         return nb_path
```

Notebook Loader

The Notebook Loader does the following three steps:

1. Load the notebook document into memory
2. Create an empty module
3. Execute every cell in the module namespace

Because IPython cells can have an extended syntax, `transform_cell` converts each cell to pure Python code before executing it.

```
[4]: class NotebookLoader(object):
    """Module Loader for IPython Notebooks"""

    def __init__(self, path=None):
        self.shell = InteractiveShell.instance()
        self.path = path

    def load_module(self, fullname):
        """import a notebook as a module"""
        path = find_notebook(fullname, self.path)

        print("importing notebook from %s" % path)

        # load the notebook object
        nb = nbformat.read(path, as_version=4)

        # create the module and add it to sys.modules
        # if name in sys.modules:
        #     return sys.modules[name]
        mod = types.ModuleType(fullname)
        mod.__file__ = path
        mod.__loader__ = self
        mod.__dict__["get_ipython"] = get_ipython
        sys.modules[fullname] = mod

        # extra work to ensure that magics that would affect the user_ns
        # magics that would affect the user_ns actually affect the
        # notebook module's ns
        save_user_ns = self.shell.user_ns
        self.shell.user_ns = mod.__dict__

        try:
            for cell in nb.cells:
                if cell.cell_type == "code":
                    # transform the input to executable Python
                    code = self.shell.input_transformer_manager.transform_cell(
                        cell.source
                    )
                    # run the code in the module
                    exec(code, mod.__dict__)
        finally:
            self.shell.user_ns = save_user_ns
        return mod
```

Notebook Finder

The Finder is a simple object that indicates whether a notebook can be imported based on its file name and that returns the appropriate loader.

```
[5]: class NotebookFinder(object):
      """Module Finder finds the transformed IPython Notebook"""

      def __init__(self):
          self.loaders = {}

      def find_module(self, fullname, path=None):
          nb_path = find_notebook(fullname, path)
          if not nb_path:
              return

          key = path
          if path:
              # lists aren't hashable
              key = os.path.sep.join(path)

          if key not in self.loaders:
              self.loaders[key] = NotebookLoader(path)
          return self.loaders[key]
```

Register hook

Now we register NotebookFinder with `sys.meta_path`:

```
[6]: sys.meta_path.append(NotebookFinder())
```

Check

Now our notebook `mypackage/foo.ipynb` should be importable with:

```
[7]: from mypackage import foo

importing notebook from /Users/veit/cusy/trn/Python4DataScience/docs/workspace/ipython/
↳ mypackage/foo.ipynb
```

Is the Python method `bar` being executed?

```
[8]: foo.bar()
```

```
[8]: 'bar'
```

... and the IPython syntax?

```
[9]: foo.dirlist()
```

```
[9]: ['debugging.ipynb',
      'display.ipynb',
      'examples.ipynb',
```

(continues on next page)

(continued from previous page)

```
'extensions.rst',
'importing.ipynb',
'index.rst',
'magics.ipynb',
'\x1b[34mmypackage\x1b[m\x1b[m',
'myscript.py',
'shell.ipynb',
'start.rst',
'\x1b[31mtab-completion-for-anything.png\x1b[m\x1b[m',
'\x1b[31mtab-completion-for-modules.png\x1b[m\x1b[m',
'\x1b[31mtab-completion-for-objects.png\x1b[m\x1b[m',
'\x1b[34munix-shell\x1b[m\x1b[m']
```

Reusable import hook

The import hook can also easily be executed in other notebooks with

```
[10]: %run display.ipynb
<IPython.core.display.HTML object>
<IPython.core.display.Javascript object>
<IPython.core.display.HTML object>
<IPython.core.display.HTML object>
```

2.1.9 IPython extensions

IPython extensions are Python modules that change the behavior of the shell. They are identified by an importable module name and are usually located in `.ipython/extensions/`.

Some important extensions are already included in IPython: `autoreload` and `storemagic`. You can find other extensions in the [Extensions Index](#) or on PyPI with the [IPython tag](#).

See also:

- [IPython extensions docs](#)

Use extensions

The `%load_ext` magic can be used to load extensions while IPython is running.

```
%load_ext myextension
```

Alternatively, an extension can also be loaded each time IPython is started by listing it in the IPython configuration file:

```
c.InteractiveShellApp.extensions = [
    'myextension'
]
```

If you haven't created an IPython configuration file yet, you can do this with:

```
$ ipython profile create [profilename]
```

If no profile name is given, default is used. The file is usually created in `~/.ipython/profile_default/` and named depending on the purpose: `ipython_config.py` is used for all IPython commands, while `ipython_notebook_config.py` is only used for commands in IPython notebooks.

Writing IPython extensions

An IPython extension is an importable Python module that has special functions for loading and unloading:

```
def load_ipython_extension(ipython):
    # The `ipython` argument is the currently active `InteractiveShell`
    # instance, which can be used in any way. This allows you to register
    # new magics or aliases, for example.

def unload_ipython_extension(ipython):
    # If you want your extension to be unloadable, put that logic here.
```

See also:

- Defining custom magics

2.1.10 Debugging

IPython contains various tools to analyse faulty code, essentially the exception reporting and the debugger.

Check exceptions with `%xmode`

If the execution of a Python script fails, an exception is usually thrown and relevant information about the cause of the error is written to a traceback. With the `%xmode` magic function you can control the amount of information that is displayed in IPython. Let's look at the following code for this:

```
[1]: def func1(a, b):
      return a / b

      def func2(x):
          a = x
          b = x - 1
          return func1(a, b)
```

```
[2]: func2(1)
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)
Cell In[2], line 1
----> 1 func2(1)

Cell In[1], line 8, in func2(x)
      6 a = x
      7 b = x - 1
```

(continues on next page)

(continued from previous page)

```
----> 8 return func1(a, b)
```

```
Cell In[1], line 2, in func1(a, b)
```

```
1 def func1(a, b):
----> 2     return a / b
```

```
ZeroDivisionError: division by zero
```

Calling `func2` leads to an error and the traceback shows exactly what happened: each line shows the context of each step that ultimately led to the error. With the `%xmode` magic function (short for exception mode) we can control which information should be displayed to us.

`%xmode` takes a single argument, the mode, and there are three options: `* Plain * Context * Verbose`

The default setting is `Context` and outputs something like the one above. `Plain` is more compact and provides less information:

```
[3]: %xmode Plain
func2(1)
```

```
Exception reporting mode: Plain
```

```
Traceback (most recent call last):
```

```
Cell In[3], line 2
    func2(1)
```

```
Cell In[1], line 8 in func2
    return func1(a, b)
Cell In[1], line 2 in func1
    return a / b
```

```
ZeroDivisionError: division by zero
```

The `Verbose` mode shows some additional information, including the arguments for any functions being called:

```
[4]: %xmode Verbose
func2(1)
```

```
Exception reporting mode: Verbose
```

```
-----
ZeroDivisionError                                Traceback (most recent call last)
```

```
Cell In[4], line 2
      1 get_ipython().run_line_magic('xmode', 'Verbose')
----> 2 func2(1)
```

```
Cell In[1], line 8, in func2(x=1)
```

```
6 a = x
7 b = x - 1
  a = 1
----> 8 return func1(a, b)
      b = 0
```

```
Cell In[1], line 2, in func1(a=1, b=0)
```

(continues on next page)

(continued from previous page)

```

1 def func1(a, b):
    a = 1
----> 2     return a / b
      b = 0

```

ZeroDivisionError: division by zero

This additional information can help narrow down the reason for the exception. Conversely, however, the **Verbose** mode can lead to extremely long tracebacks in the case of complex code, in which the essential points can hardly be recognized.

Debugging with %debug

Debugging can help if an error cannot be found by reading a traceback. The Python standard for interactive debugging is the Python debugger `pdb`. You can use it to navigate your way through the code line by line to see what is possibly causing an error. The extended version for IPython is `ipdb`.

In IPython, the `%debug`-magic command is perhaps the most convenient way to debug. If you call it after an exception has been thrown, an interactive debug prompt will automatically open during the exception. Using the `ipdb` prompt, you can examine the current status of the stack, examine the available variables and even run Python commands.

Let's look at the last exception, then do some basic tasks:

```

[5]: %debug
> /var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_21353/3792871231.
↪py(2)func1()
1 def func1(a, b):
----> 2     return a / b
      3
      4
      5 def func2(x):

ipdb> print(a)
1
ipdb> print(b)
0
ipdb> quit

```

However, the interactive debugger does a lot more – we can also go up and down the stack and examine the values of variables:

```

[6]: %debug
> /var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_21414/3792871231.
↪py(2)func1()
1 def func1(a, b):
----> 2     return a / b
      3
      4
      5 def func2(x):

ipdb> u
> /var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_21414/3792871231.

```

(continues on next page)

(continued from previous page)

```

↪py(8)func2()
4
5 def func2(x):
6     a = x
7     b = x - 1
----> 8     return func1(a, b)

ipdb> u
> /var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_21414/1541833627.py(2)
↪<module>()
1 get_ipython().run_line_magic('xmode', 'Verbose')
----> 2 func2(1)

ipdb> d
> /var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_21414/3792871231.
↪py(8)func2()
4
5 def func2(x):
6     a = x
7     b = x - 1
----> 8     return func1(a, b)

ipdb> print(x)
1
ipdb> list
3
4
5 def func2(x):
6     a = x
7     b = x - 1
----> 8     return func1(a, b)

ipdb> q

```

This greatly simplifies the search for the function calls that led to the error.

If you want the debugger to start automatically when an exception is thrown, you can use the `%pdb-magic` function to enable this behavior:

```

[7]: %xmode Plain
      %pdb on
      func2(1)

```

```

Exception reporting mode: Plain
Automatic pdb calling has been turned ON

```

```

Traceback (most recent call last):

```

```

  Cell In[7], line 3
    func2(1)

  Cell In[1], line 8 in func2
    return func1(a, b)
  Cell In[1], line 2 in func1

```

(continues on next page)

(continued from previous page)

```

    return a / b

ZeroDivisionError: division by zero

> /var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_21437/3792871231.
↪py(2)func1()
    1 def func1(a, b):
----> 2     return a / b
      3
      4
      5 def func2(x):

ipdb> p(b)
0
ipdb> q

```

If you have a script that you want to run in interactive mode from the start, you can do so with the command `%run -d`.

Essential commands of the ipdb

Command	Description
<code>list</code>	Show the current location in the file
<code>h(elp)</code>	Display a list of commands or find help on a specific command
<code>q(uit)</code>	Terminates the debugger and the program
<code>c(ontinue)</code>	Exit the debugger, continue in the program
<code>n(ext)</code>	Go to the next step in the program
<code><enter></code>	Repeat the previous command
<code>p(rint)</code>	Print variables
<code>s(tep)</code>	Step into a subroutine
<code>r(eturn)</code>	Return from a subroutine

Further information on the IPython debugger can be found at [ipdb](#).

2.2 Jupyter

We have moved the Jupyter chapter to its own tutorial: [Jupyter Tutorial](#).

2.3 NumPy

NumPy is the abbreviation for numeric Python. Many Python packages that provide scientific functions use NumPy's array objects as one of the standard interfaces for data exchange. In the following, I will give a brief overview of the main functionality of NumPy:

- *ndarray*, an efficient multidimensional array that provides fast array-based operations, such as shuffling and cleaning data, subgrouping and filtering, transformation and all other kinds of computations. There are also flexible functions for broadcasting, i.e. evaluations of arrays of different sizes.

- Mathematical functions for fast operations on whole arrays of data, such as sorting, uniqueness and set operations. Instead of loops with `if-elif-else` branches, the expressions are written in conditional logic.
- Tools for reading and writing array data to disk and working with [memory mapped](#) files.
- Functions for linear algebra, random number generation and Fourier transform.
- A C API for connecting NumPy to libraries written in C, C++ or FORTRAN.

Note: This section introduces you to the basics of using NumPy arrays and should be sufficient to follow the rest of the tutorial. For many data analytic applications, it is not necessary to have a deep understanding of NumPy, but mastering array-oriented programming and thinking is an important step on the way to becoming a data scientist.

See also:

- [Home](#)
- [Docs](#)
- [GitHub](#)
- [Tutorials](#)

2.3.1 Introduction to NumPy

NumPy operations perform complex calculations on entire arrays without the need for Python `for` loops, which can be slow for large sequences. NumPy's speed is explained by its C-based algorithms, which avoid the overhead of Python code. To give you an idea of the performance difference, we measure the difference between a NumPy array and a Python list with a hundred thousand integers:

```
[1]: import numpy as np
```

```
[2]: myarray = np.arange(100000)
     mylist = list(range(100000))
```

```
[3]: %time for _ in range(10): myarray2 = myarray ** 2
CPU times: user 2.67 ms, sys: 11.3 ms, total: 14 ms
Wall time: 1.49 ms
```

```
[4]: %time for _ in range(10): mylist2 = [x ** 2 for x in mylist]
CPU times: user 73.9 ms, sys: 320 ms, total: 394 ms
Wall time: 35.5 ms
```

2.3.2 ndarray – an N-dimensional array object

`ndarray` allows mathematical operations on whole blocks of data, using a similar syntax to similar operations between [scalar](#) elements. In NumPy, there are many different types for describing scalars, mostly based on types from the C language and those compatible with Python.

See also:

- [Array Scalars](#)

Note:

Whenever this tutorial talks about *array* or *ndarray*, in most cases it refers to the *ndarray* object.

```
[1]: import numpy as np
```

```
[2]: py_list = [2020, 2021, 2022]
      array_1d = np.array(py_list)
```

```
[3]: array_1d
```

```
[3]: array([ 2020,  2021, 2022])
```

Nested sequences, such as a list of lists of equal length, can be converted into a multidimensional array:

```
[4]: list_of_lists = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
      array_2d = np.array(list_of_lists)
```

```
[5]: array_2d
```

```
[5]: array([[ 1,  2,  3,  4],
           [ 5,  6,  7,  8],
           [ 9, 10, 11, 12]])
```

Since `list_of_lists` was a list with three lists, the NumPy array `array_2d` has two dimensions whose shape is derived from the data. With the attributes `ndim` and `shape` we can output the number of dimensions and the outline of `array_2d`:

```
[6]: array_2d.ndim
```

```
[6]: 2
```

```
[7]: array_2d.shape
```

```
[7]: (3, 4)
```

To give you an idea of the syntax, I first create an array of random numbers with five columns and seven slices:

```
[8]: data = np.random.randn(7, 3)
      data
```

```
[8]: array([[-1.48040214,  0.60483587, -0.2437932 ],
           [-0.42025594, -1.75075057,  0.19677647],
           [ 0.98816551,  0.35657111, -0.223424  ],
           [ 1.10143461,  0.25189838, -1.11756074],
           [ 0.57691653,  0.26666378,  0.68076501],
           [ 1.40382396, -0.21795603, -0.20410514],
           [ 0.64489473,  0.18392548, -0.01361532]])
```

`ndarray` is a generic multidimensional container. Each array has a shape, a tuple, which indicates the size of the individual dimensions. With `shape`, I can output the number of rows and columns in an array:

In addition to `np.array`, there are a number of other functions for creating new arrays. `zeros` and `ones`, for example, create arrays of zeros and ones, respectively, with a specific length or shape. `empty` creates an array without initialising its values to a specific value. To create a higher dimensional array using these methods, pass a tuple for the shape:


```
[9]: np.zeros(4)
```

```
[9]: array([0., 0., 0., 0.])
```

```
[10]: np.ones((3,4))
```

```
[10]: array([[1., 1., 1., 1.],
           [1., 1., 1., 1.],
           [1., 1., 1., 1.]])
```

```
[11]: np.empty((2,3,4))
```

```
[11]: array([[0., 0., 0., 0.],
           [0., 0., 0., 0.],
           [0., 0., 0., 0.]],

           [[0., 0., 0., 0.],
           [0., 0., 0., 0.],
           [0., 0., 0., 0.]])
```

Note:

You may not safely assume that the `np.empty` function returns an array of zeros, as it returns uninitialised memory and may therefore contain garbage values.

`arange` is an array-valued version of the Built-in Python `range` function:

```
[12]: np.arange(4)
```

```
[12]: array([0, 1, 2, 3])
```

Other NumPy standard functions for creating arrays are:

Function	Description
<code>array</code>	converts input data (list, tuple, array or other sequence types) into an <code>ndarray</code> by either deriving a <code>dtype</code> or explicitly specifying a <code>dtype</code> ; by default, copies the input data into the array
<code>asarray</code>	converts the input to an <code>ndarray</code> , but does not copy if the input is already an <code>ndarray</code>
<code>arange</code>	like Python built-in <code>range</code> , but returns an <code>ndarray</code> instead of a list
<code>ones</code> , <code>ones_like</code>	<code>ones</code> creates an array of 1s in the given form and <code>dtype</code> ; <code>ones_like</code> takes another array and creates an <code>ones</code> array in the same form and <code>dtype</code>
<code>zeros</code> , <code>zeros_like</code>	like <code>ones</code> and <code>ones_like</code> , but creates arrays with 0s instead
<code>empty</code> , <code>empty_like</code>	creates new arrays by allocating new memory, but does not fill them with values like <code>ones</code> and <code>zeros</code>
<code>full</code> , <code>full_like</code>	creates an array of the given <code>shape</code> and <code>dtype</code> , setting all values to the given fill value; <code>full_like</code> takes another array and creates a filled array with the same <code>shape</code> and <code>dtype</code>
<code>eye</code> , <code>identity</code>	creates a square $N \times N$ identity matrix (1s on the diagonal and 0s elsewhere)

2.3.3 dtype

`ndarray` is a container for homogeneous data, i.e. all elements must be of the same type. Each array has a `dtype`, an object that describes the data type of the array:

```
[1]: import numpy as np
```

```
data = np.random.randn(7, 3)
dt = data.dtype
dt
```

```
[1]: dtype('float64')
```

NumPy data types:

Type	Type code	Description
<code>int8, uint8</code>	<code>i1, u1</code>	Signed and unsigned 8-bit (1-byte) integer types
<code>int16, uint16</code>	<code>i2, u2</code>	Signed and unsigned 16-Bit (2 Byte) integer types
<code>int32, uint32</code>	<code>i4, u4</code>	Signed and unsigned 32-Bit (4 Byte) integer types
<code>int64, uint64</code>	<code>i8, u8</code>	Signed and unsigned 64-Bit (8 Byte) integer types
<code>float16</code>	<code>f2</code>	Standard floating point with half precision
<code>float32</code>	<code>f4</code> or <code>f</code>	Standard floating point with single precision; compatible with C float
<code>float64</code>	<code>f8</code> or <code>d</code>	Standard floating point with double precision; compatible with C double and Python float object
<code>complex64, complex128, complex256</code>	<code>c8, c16, c32</code>	Complex numbers represented by two 32, 64 or 128 floating point numbers respectively
<code>bool</code>	<code>?</code>	Boolean type that stores the values <code>True</code> and <code>False</code>
<code>object</code>	<code>0</code>	Python object type; a value can be any Python object
<code>string_</code>	<code>S</code>	ASCII string type with fixed length (1 byte per character); to create a string type with length 7, for example, use <code>S7</code> ; longer inputs are truncated without warning
<code>unicode_</code>	<code>U</code>	Unicode type with fixed length where the number of bytes is platform-specific; uses the same specification semantics as <code>string_</code> , e.g. <code>U7</code>

Determine the number of elements with `itemsize`:

```
[2]: dt.itemsize
```

```
[2]: 8
```

Determine the name of the data type:

```
[3]: dt.name
```

```
[3]: 'float64'
```

Check data type:

```
[4]: dt.type is np.float64
```

```
[4]: True
```

Change data type with `astype`:

```
[5]: data_float32 = data.astype(np.float32)
data_float32

[5]: array([[ 0.12408408,  0.28413823,  1.6867595 ],
          [-0.4144261 , -0.5990565 ,  0.61371785],
          [ 0.16093737,  0.12486719, -0.16383053],
          [ 1.0395902 , -1.4354634 ,  0.35893318],
          [ 0.82148165, -2.134709 ,  0.12962751],
          [-1.0212289 ,  0.72899795, -1.7471288 ],
          [-1.8143699 , -1.0880227 , -1.1238078 ]], dtype=float32)
```

2.3.4 Arithmetic

Arrays allow you to perform stack operations on data without having to use `for` loops. This is called *vectorisation* in NumPy. For all arithmetic operations between arrays of the same size, the operation is applied element by element:

```
[1]: import numpy as np

data = np.random.randn(7, 3)
data

[1]: array([[ -0.52169857, -0.06638825, -1.70235417],
          [  0.3540172 , -1.30560063, -1.0368024 ],
          [-0.4163764 , -1.24874081, -1.85063163],
          [-0.63982944, -0.47325691,  1.42545299],
          [  1.11960638,  1.49821503, -0.11843174],
          [-0.59220784,  0.63391355,  1.21890647],
          [-0.57770878,  1.05719525,  2.54019148]])

[2]: 1 / data

[2]: array([[ -1.91681569, -15.06290747, -0.58742183],
          [  2.82472155, -0.765931 , -0.96450394],
          [-2.40167311, -0.8008067 , -0.54035605],
          [-1.56291653, -2.11301722,  0.70153138],
          [  0.89317104,  0.66746093, -8.44368223],
          [-1.68859637,  1.57750217,  0.82040749],
          [-1.73097595,  0.94589907,  0.39367111]])

[3]: data**2

[3]: array([[2.72169395e-01, 4.40739915e-03, 2.89800972e+00],
          [1.25328175e-01, 1.70459301e+00, 1.07495921e+00],
          [1.73369306e-01, 1.55935360e+00, 3.42483742e+00],
          [4.09381707e-01, 2.23972103e-01, 2.03191622e+00],
          [1.25351846e+00, 2.24464828e+00, 1.40260776e+02],
          [3.50710120e-01, 4.01846387e-01, 1.48573298e+00],
          [3.33747430e-01, 1.11766179e+00, 6.45257275e+00]])
```

Comparison of two arrays:

```
[4]: data2 = np.random.randn(7, 3)
data > data2

[4]: array([[False, False, False],
          [False, False, False],
          [False, False, False],
          [ True,  True,  True],
          [ True,  True, False],
          [False,  True,  True],
          [False,  True,  True]])
```

2.3.5 Indexing and slicing

Indexing is the selection of a subset of your data or individual elements. This is very easy in one-dimensional arrays; they behave similarly to Python lists:

```
[1]: import numpy as np

[2]: rng = np.random.default_rng()
data = rng.normal(size=(10, 3))
data

[2]: array([[ -0.1781624 , -0.8381147 ,  1.40248986],
          [-1.48367758,  0.70035394,  0.60506565],
          [ 2.24316514,  0.38021158,  0.95148769],
          [-0.37414371,  1.03258406, -1.51360252],
          [-1.6251526 ,  0.34516475,  0.6205052 ],
          [ 0.96867556,  0.13047506, -1.80399701],
          [-0.20605706, -1.04783043,  0.69553167],
          [ 1.14186171, -1.01894781, -1.44487713],
          [ 0.29214215,  1.60380789, -1.82980606],
          [-1.87650688, -0.5427789 ,  1.6327612 ]])

[3]: data[4]

[3]: array([-1.6251526 ,  0.34516475,  0.6205052 ])

[4]: data[2:4]

[4]: array([[ 2.24316514,  0.38021158,  0.95148769],
          [-0.37414371,  1.03258406, -1.51360252]])

[5]: data[2:4] = rng.normal(size=(2, 3))

[6]: data

[6]: array([[ -0.1781624 , -0.8381147 ,  1.40248986],
          [-1.48367758,  0.70035394,  0.60506565],
          [-0.07210875, -0.4775101 , -1.09241001],
          [ 2.45845089, -0.26972796, -2.0442523 ],
          [-1.6251526 ,  0.34516475,  0.6205052 ],
          [ 0.96867556,  0.13047506, -1.80399701],
```

(continues on next page)

(continued from previous page)

```
[-0.20605706, -1.04783043,  0.69553167],
[ 1.14186171, -1.01894781, -1.44487713],
[ 0.29214215,  1.60380789, -1.82980606],
[-1.87650688, -0.5427789 ,  1.6327612  ]])
```

Note:

Array slices differ from Python lists in that they are views of the original array. This means that the data is not copied and that any changes to the view are reflected in the original array.

If you want to make a copy of a part of an `ndarray`, you can copy the array explicitly – for example with `data[2:5].copy()`.

Slicing in this way always results in array views with the same number of dimensions. However, if you mix integer indices and slices, you get slices with lower dimensions. For example, we can select the second row but only the first two columns as follows:

```
[7]: data[1, :2]
[7]: array([-1.48367758,  0.70035394])
```

A colon means that the whole axis is taken, so you can also select higher dimensional axes:

```
[8]: data[:, :1]
[8]: array([[ -0.1781624 ],
           [-1.48367758],
           [-0.07210875],
           [ 2.45845089],
           [-1.6251526 ],
           [ 0.96867556],
           [-0.20605706],
           [ 1.14186171],
           [ 0.29214215],
           [-1.87650688]])
```

Boolean indexing

Let's consider an example where we have some data in an array and an array of names with duplicates. I will use the `normal` function in `numpy.random.default_rng` here to generate some random normally distributed data:

```
[9]: names = np.array(
    [
        "Liam",
        "Olivia",
        "Noah",
        "Liam",
        "Noah",
        "Olivia",
        "Liam",
        "Emma",
        "Oliver",
        "Ava",
```

(continues on next page)

(continued from previous page)

```
]
)
```

```
[10]: names
```

```
[10]: array(['Liam', 'Olivia', 'Noah', 'Liam', 'Noah', 'Olivia', 'Liam', 'Emma',
          'Oliver', 'Ava'], dtype='<U6')
```

```
[11]: data
```

```
[11]: array([[ -0.1781624 , -0.8381147 ,  1.40248986],
             [-1.48367758,  0.70035394,  0.60506565],
             [-0.07210875, -0.4775101 , -1.09241001],
             [ 2.45845089, -0.26972796, -2.0442523 ],
             [-1.6251526 ,  0.34516475,  0.6205052 ],
             [ 0.96867556,  0.13047506, -1.80399701],
             [-0.20605706, -1.04783043,  0.69553167],
             [ 1.14186171, -1.01894781, -1.44487713],
             [ 0.29214215,  1.60380789, -1.82980606],
             [-1.87650688, -0.5427789 ,  1.6327612 ]])
```

Suppose each name corresponds to a row in the data array and we want to select all rows with the corresponding name *Liam*. Like arithmetic operations, comparisons like `==` are vectorised with arrays. So comparing names with the string *Liam* results in a Boolean array:

```
[12]: names == "Liam"
```

```
[12]: array([ True, False, False,  True, False, False,  True, False, False,
          False])
```

This Boolean array can be passed when indexing the array:

```
[13]: data[names == "Liam"]
```

```
[13]: array([[ -0.1781624 , -0.8381147 ,  1.40248986],
             [ 2.45845089, -0.26972796, -2.0442523 ],
             [-0.20605706, -1.04783043,  0.69553167]])
```

Here, the Boolean array must have the same length as the array axis it indexes.

Note:

Selecting data from an array by Boolean indexing and assigning the result to a new variable always creates a copy of the data, even if the returned array is unchanged.

In the following example, I select the rows where `names == 'Liam'` and also index the columns:

```
[14]: data[names == "Liam", 2:]
```

```
[14]: array([[ 1.40248986],
             [-2.0442523 ],
             [ 0.69553167]])
```

To select everything except *Liam*, you can either use `!=` or negate the condition with `~`:

```
[15]: names != "Liam"
```

```
[15]: array([False,  True,  True, False,  True,  True, False,  True,  True,
           True])
```

```
[16]: cond = names == "Liam"
      data[~cond]
```

```
[16]: array([[ -1.48367758,  0.70035394,  0.60506565],
             [-0.07210875, -0.4775101 , -1.09241001],
             [-1.6251526 ,  0.34516475,  0.6205052 ],
             [ 0.96867556,  0.13047506, -1.80399701],
             [ 1.14186171, -1.01894781, -1.44487713],
             [ 0.29214215,  1.60380789, -1.82980606],
             [-1.87650688, -0.5427789 ,  1.6327612 ]])
```

If you select two of the three names to combine several Boolean conditions, you can use the Boolean arithmetic operators & (and) and | (or).

Warning:

The Python keywords `and` and `or` do not work with Boolean arrays.

```
[17]: mask = (names == "Liam") | (names == "Olivia")
```

```
[18]: mask
```

```
[18]: array([ True,  True, False,  True, False,  True,  True, False, False,
           False])
```

```
[19]: data[mask]
```

```
[19]: array([[ -0.1781624 , -0.8381147 ,  1.40248986],
             [-1.48367758,  0.70035394,  0.60506565],
             [ 2.45845089, -0.26972796, -2.0442523 ],
             [ 0.96867556,  0.13047506, -1.80399701],
             [-0.20605706, -1.04783043,  0.69553167]])
```

Integer Array Indexing

Integer array indexing allows you to select any elements in the array based on your N-dimensional index. Each integer array represents a number of indices in that dimension.

See also:

- [Integer array indexing](#)

2.3.6 Transpose arrays and swap axes

Transpose is a special form of reshaping that also provides a view of the underlying data without copying anything. Arrays have the `Transpose` method and also the special `T` attribute:

```
[1]: import numpy as np

[2]: data = np.arange(16)

[3]: data
[3]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])

[4]: reshaped_data = data.reshape((4, 4))

[5]: reshaped_data
[5]: array([[ 0,  1,  2,  3],
          [ 4,  5,  6,  7],
          [ 8,  9, 10, 11],
          [12, 13, 14, 15]])

[6]: reshaped_data.T
[6]: array([[ 0,  4,  8, 12],
          [ 1,  5,  9, 13],
          [ 2,  6, 10, 14],
          [ 3,  7, 11, 15]])
```

`numpy.dot` returns the scalar product of two arrays, for example:

```
[7]: np.dot(reshaped_data.T, reshaped_data)
[7]: array([[224, 248, 272, 296],
          [248, 276, 304, 332],
          [272, 304, 336, 368],
          [296, 332, 368, 404]])
```

The `@` infix operator is another way to perform matrix multiplication. It implements the semantics of the `@` operator introduced in Python 3.5 with [PEP 465](#) and is an abbreviation of `np.matmul`.

```
[8]: data.T @ data
[8]: 1240
```

For higher dimensional arrays, `transpose` accepts a tuple of axis numbers to swap the axes:

```
[9]: array_3d = np.arange(16).reshape((2, 2, 4))

[10]: array_3d
[10]: array([[[ 0,  1,  2,  3],
            [ 4,  5,  6,  7]],
          [[ 8,  9, 10, 11],
            [12, 13, 14, 15]])])
```



```
[11]: array_3d.transpose((1, 0, 2))
```

```
[11]: array([[[ 0,  1,  2,  3],
              [ 8,  9, 10, 11]],

            [[ 4,  5,  6,  7],
             [12, 13, 14, 15]]])
```

Here the axes have been reordered with the second axis in first place, the first axis in second place and the last axis unchanged.

`ndarray` also has a `swapaxes` method that takes a pair of axis numbers and swaps the specified axes to rearrange the data:

```
[12]: array_3d.swapaxes(1, 2)
```

```
[12]: array([[[ 0,  4],
              [ 1,  5],
              [ 2,  6],
              [ 3,  7]],

            [[ 8, 12],
             [ 9, 13],
             [10, 14],
             [11, 15]])])
```

2.3.7 Universal functions (ufunc)

A universal function, or `ufunc`, is a function that performs element-wise operations on data in `ndarrays`. They can be thought of as fast vectorised wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

Many `ufuncs` are simple element-wise transformations, such as `sqrt` or `exp`:

```
[1]: import numpy as np
```

```
data = np.arange(10)
```

```
[2]: data
```

```
[2]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
[3]: np.sqrt(data)
```

```
[3]: array([0.          , 1.          , 1.41421356, 1.73205081, 2.          ,
           2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.          ])
```

```
[4]: np.exp(data)
```

```
[4]: array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
           5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
           2.98095799e+03, 8.10308393e+03])
```

These are called single-digit ufuncs. Others, such as `add` or `maximum`, take two arrays (i.e. binary ufuncs) and return a single array as the result:

```
[5]: x = np.random.randn(8)
```

```
[6]: y = np.random.randn(8)
```

```
[7]: x
```

```
[7]: array([-1.23545026, -2.97614783,  1.81553171,  1.01874633,  0.08063104,  
        -0.4605132 ,  2.26014706,  1.88403856])
```

```
[8]: y
```

```
[8]: array([ 0.70506547, -0.64166724,  2.10440297,  0.09330584, -1.47706135,  
        -0.99220346,  0.54688573,  0.06453598])
```

```
[9]: np.maximum(x, y)
```

```
[9]: array([ 0.70506547, -0.64166724,  2.10440297,  1.01874633,  0.08063104,  
        -0.4605132 ,  2.26014706,  1.88403856])
```

Here `numpy.maximum` calculated the element-wise maximum of the elements in `x` and `y`.

Some ufunc, such as `modf`, a vectorised version of the built-in Python `divmod`, return multiple arrays: the fractional and integral parts of a floating-point array:

```
[10]: data = x * 5
```

```
[11]: data
```

```
[11]: array([-6.1772513 , -14.88073913,  9.07765855,  5.09373163,  
         0.40315522, -2.30256598, 11.3007353 ,  9.42019279])
```

```
[12]: remainder, whole_part = np.modf(x)
```

```
[13]: remainder
```

```
[13]: array([-0.23545026, -0.97614783,  0.81553171,  0.01874633,  0.08063104,  
        -0.4605132 ,  0.26014706,  0.88403856])
```

```
[14]: whole_part
```

```
[14]: array([-1., -2.,  1.,  1.,  0., -0.,  2.,  1.])
```

Ufuncs accept an optional `out` argument that allows you to transfer your results to an existing array instead of creating a new one:

```
[15]: out = np.zeros_like(data)
```

```
[16]: np.add(data, 1)
```

```
[16]: array([-5.1772513 , -13.88073913, 10.07765855,  6.09373163,  
         1.40315522, -1.30256598, 12.3007353 , 10.42019279])
```

```
[17]: np.add(data, 1, out=out)
```

```
[17]: array([ -5.1772513 , -13.88073913,  10.07765855,   6.09373163,
          1.40315522, -1.30256598,  12.3007353 ,  10.42019279])
```

```
[18]: out
```

```
[18]: array([ -5.1772513 , -13.88073913,  10.07765855,   6.09373163,
          1.40315522, -1.30256598,  12.3007353 ,  10.42019279])
```

Some single-digit ufuncs:

Function	Description
abs, fabs	calculates the absolute value element by element for integer, floating point or complex values
sqrt	calculates the square root of each element (corresponds to <code>data ** 0.5</code>)
square	calculates the square of each element (corresponds to <code>data ** 2</code>)
exp	calculates the exponent <code>e^x</code> of each element
log, log10, log2, log1p	Natural logarithm (base <code>e</code>), log base 10, log base 2 and <code>log(1 + x)</code> respectively
sign	calculates the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
ceil	calculates the upper limit of each element (i.e. the smallest integer greater than or equal to this number)
floor	calculates the lower limit of each element (i.e. the largest integer less than or equal to each element)
rint	rounds elements to the nearest integer, preserving the dtype
modf	returns the fractional and integer parts of the array as separate arrays
isnan	returns a boolean array indicating whether each value is NaN (Not a Number)
isfinite, isinf	returns a boolean array indicating whether each element is finite (not-inf, not-NaN) or infinite, respectively
cos, cosh, sin, sinh, tan, tanh	regular and hyperbolic trigonometric functions
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Inverse trigonometric functions
logical_not	calculates the truth value of <code>not x</code> element by element (corresponds to <code>~data</code>)

Some binary universal functions:

Function	Description
<code>add</code>	add corresponding elements in arrays
<code>subtract</code>	subtracts elements in the second array from the first array
<code>multiply</code>	multiply array elements
<code>divide, floor_divide</code>	divide or truncate the remainder
<code>power</code>	increases elements in the first array to the powers specified in the second array
<code>maximum, fmax</code>	element-wise maximum; <code>fmax</code> ignores NaN
<code>minimum, fmin</code>	element-wise minimum; <code>fmin</code> ignores NaN
<code>mod</code>	element-wise modulus (remainder of the division)
<code>copysign</code>	copies the sign of the values in the second argument to the values in the first argument
<code>greater, greater_equal, less, less_equal, equal, not_equal</code>	perform element-wise comparisons that result in a Boolean array (corresponds to the infix operators <code>></code> , <code>>=</code> , <code><</code> , <code><=</code> , <code>==</code> , <code>!=</code>)
<code>logical_and</code>	calculates the element-wise truth value of the logical operation AND (<code>&</code>)
<code>logical_or</code>	calculates the element-wise truth value of the logical operation OR (<code> </code>)
<code>logical_xor</code>	calculates the element-wise truth value of the logical operation XOR (<code>^</code>)

Note:

A complete overview of binary universal functions can be found in [Universal functions \(ufunc\)](#).

2.3.8 Array-oriented programming – vectorisation

Using NumPy arrays allows you to express many types of data processing tasks as concise array expressions that would otherwise require writing `for`-loops. This practice of replacing loops with array expressions is also called *vectorisation*. In general, vectorised array operations are significantly faster than their pure Python equivalents.

```
[1]: import numpy as np
```

First we create a NumPy array with one hundred thousand integers:

```
[2]: myarray = np.arange(100000)
```

Then we square all the elements in this array with `numpy.square`:

```
[3]: %time np.square(myarray)
```

```
CPU times: user 559 µs, sys: 2.55 ms, total: 3.11 ms
Wall time: 269 µs
```

```
[3]: array([      0,         1,         4, ..., 9999400009, 9999600004,
          9999800001])
```

For comparison, we now measure the time of Python's quadratic function:

```
[4]: %time for _ in range(10): myarray2 = myarray ** 2
```

```
CPU times: user 807 µs, sys: 4.07 ms, total: 4.87 ms
Wall time: 440 µs
```

And finally, we compare the time with the calculation of the quadratic function of all values of a Python list:

```
[5]: mylist = list(range(100000))
      %time for _ in range(10): mylist2 = [x ** 2 for x in mylist]

CPU times: user 115 ms, sys: 390 ms, total: 505 ms
Wall time: 46.7 ms
```

2.3.9 Conditional logic as array operations – where

The `numpy.where` function is a vectorised version of `if` and `else`.

In the following example, we first create a Boolean array and two arrays with values:

```
[1]: import numpy as np

[2]: cond = ([False, True, False, True, False, False, False])
      data1 = np.random.randn(1, 7)
      data2 = np.random.randn(1, 7)
```

Now we want to take the values from `data1` if the corresponding value in `cond` is `True` and otherwise take the value from `data2`. With Python's `if-else`, this could look like this:

```
[3]: result = [(x if c else y) for x, y, c in zip(data1, data2, cond)]

result

[3]: [array([ 0.0753595 ,  0.70598847,  1.36375888,  0.52613878,  1.58394917,
            -0.67041886, -1.30890145])]
```

However, this has the following two problems:

- with large arrays the function will not be very fast
- this will not work with multidimensional arrays

With `np.where` you can work around these problems in a single function call:

```
[4]: result = np.where(cond, data1, data2)

result

[4]: array([[ 0.0753595 , -0.97727968,  1.36375888,  1.5042741 ,  1.58394917,
            -0.67041886, -1.30890145]])
```

The second and third arguments of `np.where` do not have to be arrays; one or both can also be scalars. A typical use of `where` in data analysis is to create a new array of values based on another array. Suppose you have a matrix of randomly generated data and you want to make all the negative values positive values:

```
[5]: data = np.random.randn(4, 4)

data

[5]: array([[ -2.13569944,  0.21406577, -0.44948598,  0.07841356],
           [ 0.94045485, -0.47748714, -0.70057099, -1.92553004],
           [-1.65814642,  0.44475682, -1.16289192,  0.96023582],
           [ 0.45396769,  0.64944133, -0.08936879, -1.20179191]])
```

```
[6]: data < 0
[6]: array([[ True, False,  True, False],
          [False,  True,  True,  True],
          [ True, False,  True, False],
          [False, False,  True,  True]])

[7]: np.where(data < 0, data * -1, data)
[7]: array([[2.13569944, 0.21406577, 0.44948598, 0.07841356],
          [0.94045485, 0.47748714, 0.70057099, 1.92553004],
          [1.65814642, 0.44475682, 1.16289192, 0.96023582],
          [0.45396769, 0.64944133, 0.08936879, 1.20179191]])
```

2.3.10 Mathematical and statistical methods

A number of mathematical functions that calculate statistics over an entire array or over the data along an axis are accessible as methods of the array class. So you can use aggregations such as sum, mean and standard deviation by either calling the array instance method or using the top-level NumPy function.

Below I generate some random data and calculate some aggregated statistics:

```
[1]: import numpy as np

data = np.random.randn(7, 3)

data
[1]: array([[ 0.52892401, -0.82705139, -0.13426779],
          [-0.43476595,  0.15431376, -0.15927356],
          [ 0.5437757 , -0.27273503, -0.74511308],
          [ 0.41921053,  0.78804831, -1.39898524],
          [-0.08745354,  0.24346498,  0.5995653 ],
          [ 2.18987033,  0.07709088,  0.81486999],
          [ 0.42570339,  1.23702332,  1.12807273]])

[2]: data.mean()
[2]: 0.24239465071821545

[3]: np.mean(data)
[3]: 0.24239465071821545

[4]: data.sum()
[4]: 5.090287665082524
```

Functions like mean and sum require an optional axis argument that calculates the statistic over the specified axis, resulting in an array with one less dimension:

```
[5]: data.mean(axis=0)
```

```
[5]: array([0.51218064, 0.20002212, 0.01498119])
```

```
[6]: data.sum(axis=0)
```

```
[6]: array([3.58526448, 1.40015484, 0.10486835])
```

With `data.mean(0)`, which is the same as `data.mean(axis=0)`, the mean is calculated over the rows, while `data.sum(0)` calculates the sum over the rows.

Other methods like `cumsum` and `cumprod`, however, do not aggregate but create a new array with the intermediate results.

In multidimensional arrays, accumulation functions such as `cumsum` and `cumprod` return an array of the same size but with the partial aggregates calculated along the specified axis:

```
[7]: data.cumsum()
```

```
[7]: array([ 0.52892401, -0.29812737, -0.43239516, -0.86716111, -0.71284735,
          -0.87212091, -0.32834522, -0.60108025, -1.34619332, -0.92698279,
          -0.13893449, -1.53791972, -1.62537326, -1.38190829, -0.78234299,
           1.40752735,  1.48461823,  2.29948822,  2.72519162,  3.96221494,
           5.09028767])
```

```
[8]: data.cumprod()
```

```
[8]: array([ 5.28924012e-01, -4.37447338e-01,  5.87350864e-02, -2.55360156e-02,
          -3.94055863e-03,  6.27626816e-04,  3.41288209e-04, -9.30812494e-05,
           6.93560562e-05,  2.90747892e-05,  2.29123384e-05, -3.20540232e-05,
           2.80323775e-06,  6.82490215e-07,  4.09197451e-07,  8.96089358e-07,
           6.90803200e-08,  5.62914796e-08,  2.39634740e-08,  2.96433762e-08,
           3.34398842e-08])
```

Basic statistical methods for arrays are:

Method	Description
<code>sum</code>	Sum of all elements in the array or along an axis.
<code>mean</code>	Arithmetic mean; for arrays with length zero, NaN is returned.
<code>std, var</code>	Standard deviation and variance respectively
<code>min, max</code>	Minimum and maximum
<code>argmin, argmax</code>	Indices of the minimum and maximum elements respectively
<code>cumsum</code>	Cumulative sum of the elements, starting with 0
<code>cumprod</code>	Cumulative product of the elements, starting with 1

2.3.11 Methods for Boolean arrays

Boolean values have been converted to 1 (True) and 0 (False) in the previous methods. Therefore, `sum` is often used to count the True values in a Boolean array:

```
[1]: import numpy as np
```

```
[2]: data = np.random.randn(7, 3)
```

Number of positive values:

```
[3]: (data > 0).sum()
```

```
[3]: 12
```

Number of negative values:

```
[4]: (data < 0).sum()
```

```
[4]: 9
```

There are two additional methods, `any` and `all`, which are particularly useful for Boolean arrays:

- `any` checks whether one or more values in an array are true
- `all` checks whether each value is true

```
[5]: data2 = np.random.randn(7, 3)
```

```
bools = data > data2
```

```
bools
```

```
[5]: array([[ True,  True,  True],
          [ True,  True, False],
          [ True, False, False],
          [ True, False,  True],
          [ True,  True, False],
          [False, False,  True],
          [False, False, False]])
```

```
[6]: bools.any()
```

```
[6]: True
```

```
[7]: bools.all()
```

```
[7]: False
```

2.3.12 Sort

As in Python's list, NumPy arrays can be sorted in-place using the `numpy.sort` method. You can sort any one-dimensional section of values in a multidimensional array in place along an axis by passing the axis number to sort:

```
[1]: import numpy as np
```

```
data = np.random.randn(7, 3)
```

```
data
```

```
[1]: array([[ -0.50687148, -0.92123541, -1.33470444],
          [-0.47316782, -0.05354427,  0.3144167 ],
          [-0.51270165, -1.30401598, -0.9362869 ],
          [-0.19429791,  1.12032183,  0.19184738],
          [ 0.07609175,  1.75052865, -1.27389361],
```

(continues on next page)

(continued from previous page)

```
[ 1.03374626, -0.29737004,  0.0944219 ],
[ 0.82837672, -0.29511481, -0.25849806]])
```

```
[2]: data.sort(0)
```

```
data
```

```
[2]: array([[ -0.51270165, -1.30401598, -1.33470444],
          [ -0.50687148, -0.92123541, -1.27389361],
          [ -0.47316782, -0.29737004, -0.9362869 ],
          [ -0.19429791, -0.29511481, -0.25849806],
          [  0.07609175, -0.05354427,  0.0944219 ],
          [  0.82837672,  1.12032183,  0.19184738],
          [  1.03374626,  1.75052865,  0.3144167 ]])
```

`np.sort`, on the other hand, returns a sorted copy of an array instead of changing the array in place:

```
[3]: np.sort(data, axis=1)
```

```
[3]: array([[ -1.33470444, -1.30401598, -0.51270165],
          [ -1.27389361, -0.92123541, -0.50687148],
          [ -0.9362869 , -0.47316782, -0.29737004],
          [ -0.29511481, -0.25849806, -0.19429791],
          [ -0.05354427,  0.07609175,  0.0944219 ],
          [  0.19184738,  0.82837672,  1.12032183],
          [  0.3144167 ,  1.03374626,  1.75052865]])
```

```
[4]: data
```

```
[4]: array([[ -0.51270165, -1.30401598, -1.33470444],
          [ -0.50687148, -0.92123541, -1.27389361],
          [ -0.47316782, -0.29737004, -0.9362869 ],
          [ -0.19429791, -0.29511481, -0.25849806],
          [  0.07609175, -0.05354427,  0.0944219 ],
          [  0.82837672,  1.12032183,  0.19184738],
          [  1.03374626,  1.75052865,  0.3144167 ]])
```

2.3.13 unique and other set logic

NumPy has some basic set operations for one-dimensional `ndarray`. A commonly used one is `numpy.unique`, which returns the sorted unique values in an array:

```
[1]: import numpy as np
```

```
names = np.array(
    [
        "Liam",
        "Olivia",
        "Noah",
        "Liam",
        "Noah",
```

(continues on next page)

(continued from previous page)

```

        "Olivia",
        "Liam",
        "Emma",
        "Oliver",
        "Ava",
    ]
)

```

```
[2]: np.unique(names)
```

```
[2]: array(['Ava', 'Emma', 'Liam', 'Noah', 'Oliver', 'Olivia'], dtype='<U6')
```

With `numpy.in1d` you can check the membership of the values in a one-dimensional array to another array and a boolean array is returned:

```
[3]: np.in1d(names, ["Emma", "Ava", "Charlotte"])
```

```
[3]: array([False, False, False, False, False, False, False,  True, False,
           True])
```

Array set operations:

Method	Description
<code>unique(x)</code>	calculates the sorted, unique elements in <code>x</code>
<code>intersect1d(x, y)</code>	calculates the sorted common elements <code>x</code> and <code>y</code>
<code>union1d(x, y)</code>	calculates the sorted union of elements
<code>in1d(x, y)</code>	computes a boolean array indicating whether each element of <code>x</code> is contained in <code>y</code>
<code>setdiff1d(x, y)</code>	sets the difference of the elements in <code>x</code> that are not contained in <code>y</code>
<code>setxor1d(x, y)</code>	sets symmetric differences; elements contained in one of the arrays but not in both

2.3.14 File input and output with arrays

NumPy is able to store data in some text or binary formats on disk and load it from there. However, in this section I only discuss NumPy's own binary format, as mostly pandas or other tools are used to load text or table data (see [Read, persist and provide data](#)).

`np.save` and `np.load` are the two most important functions for efficiently saving and loading array data to disk. Arrays are saved by default in an uncompressed raw binary format with the file extension `.npy`:

```
[1]: import numpy as np
```

```
data = np.random.randn(7, 3)
```

```
np.save("my_data", data)
```

If the file path does not already end in `.npy`, the extension is appended. The array on the hard disk can then be loaded with `np.load`:

```
[2]: np.load("my_data.npy")
```

```
[2]: array([[ 1.71143962,  1.06249012,  0.40089528],
          [-1.93836029,  0.60398033, -0.6708609 ],
          [ 0.24042536, -0.86181626,  0.33594052],
          [-1.41716277,  2.11203343, -0.09469748],
          [-0.36027506,  0.53376748,  1.302226  ],
          [ 0.24560584,  1.29705793,  0.49696571],
          [ 0.04375581,  0.88412494, -2.22439157]])
```

You can save multiple arrays in an uncompressed archive by using `np.savez` and passing the arrays as keyword arguments:

```
[3]: np.savez("data_archive.npz", a=data, b=np.square(data))
```

```
[4]: archive = np.load("data_archive.npz")
```

```
archive["b"]
```

```
[4]: array([[2.92902558e+00, 1.12888526e+00, 1.60717029e-01],
          [3.75724062e+00, 3.64792237e-01, 4.50054349e-01],
          [5.78043555e-02, 7.42727271e-01, 1.12856032e-01],
          [2.00835032e+00, 4.46068522e+00, 8.96761189e-03],
          [1.29798116e-01, 2.84907727e-01, 1.69579255e+00],
          [6.03222306e-02, 1.68235927e+00, 2.46974919e-01],
          [1.91457098e-03, 7.81676918e-01, 4.94791787e+00]])
```

2.4 pandas

`pandas` is a Python library for data analysis that has become very popular in recent years. On the website, `pandas` is described thus:

„`pandas` is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.“

More specifically, `pandas` is an in-memory analysis tool that offers SQL-like constructs, as well as statistical and analytical tools. In doing so, `pandas` builds on Cython and NumPy, making it less memory intensive and faster than pure Python code. Mostly `pandas` is used to

- replace *Excel* and *Power BI*
- implement an *ETL* process
- process *CSV* or *JSON* data
- prepare machine learning

See also:

- [Home](#)
- [User guide](#)
- [API reference](#)
- [GitHub](#)

2.4.1 Introduction to the data structures of pandas

To get started with pandas, you should first familiarise yourself with the two most important data structures *Series* and *DataFrame*.

Series

A series is a one-dimensional array-like object containing a sequence of values (of similar types to the NumPy types) and an associated array of data labels called an index. The simplest series is formed from just an array of data:

```
[1]: import numpy as np
import pandas as pd

[2]: rng = np.random.default_rng()
s = pd.Series(rng.normal(size=7))
s

[2]: 0    0.415497
1    3.102087
2   -0.332863
3   -0.135429
4    0.471112
5    0.173483
6   -0.487151
dtype: float64
```

The string representation of an interactively displayed series shows the index on the left and the values on the right. Since we have not specified an index for the data, a default index is created consisting of the integers 0 to $N - 1$ (where N is the length of the data). You can get the array representation and the index object of the series via their `pandas.Series.array` and `pandas.Series.index` attributes respectively:

```
[3]: s.array
[3]: <PandasArray>
[ 0.4154969051865909,    3.102087203833539,   -0.3328632996406089,
 -0.13542859429409687,    0.4711123318607415,    0.1734826179409076,
 -0.48715121240065956]
Length: 7, dtype: float64

[4]: s.index
[4]: RangeIndex(start=0, stop=7, step=1)
```

Often you will want to create an index that identifies each data point with a label:

```
[5]: idx = pd.date_range("2022-01-31", periods=7)
s2 = pd.Series(rng.normal(size=7), index=idx)

[6]: s2
[6]: 2022-01-31    0.434474
2022-02-01   -1.696645
2022-02-02   -1.180240
```

(continues on next page)

(continued from previous page)

```

2022-02-03    -0.205702
2022-02-04    -0.426140
2022-02-05    -0.123695
2022-02-06     1.071786
Freq: D, dtype: float64

```

See also:

- [Time series / date functionality](#)

Compared to NumPy arrays, you can use labels in the index if you want to select individual values or a group of values:

```
[7]: s2["2022-02-02"]
```

```
[7]: -1.1802398819304771
```

```
[8]: s2[["2022-02-02", "2022-02-03", "2022-02-04"]]
```

```
[8]: 2022-02-02    -1.180240
      2022-02-03    -0.205702
      2022-02-04    -0.426140
      dtype: float64

```

Here `['2022-02-02', '2022-02-03', '2022-02-04']` is interpreted as a list of indices, even if it contains strings instead of integers.

When using NumPy functions or NumPy-like operations, such as filtering with a Boolean array, scalar multiplication or applying mathematical functions, the link between index and value is preserved:

```
[9]: s2[s2 > 0]
```

```
[9]: 2022-01-31     0.434474
      2022-02-06     1.071786
      dtype: float64

```

```
[10]: s2**2
```

```
[10]: 2022-01-31     0.188768
      2022-02-01     2.878604
      2022-02-02     1.392966
      2022-02-03     0.042313
      2022-02-04     0.181595
      2022-02-05     0.015301
      2022-02-06     1.148725
      Freq: D, dtype: float64

```

```
[11]: np.exp(s2)
```

```
[11]: 2022-01-31     1.544151
      2022-02-01     0.183297
      2022-02-02     0.307205
      2022-02-03     0.814076
      2022-02-04     0.653025
      2022-02-05     0.883649
      2022-02-06     2.920591
      Freq: D, dtype: float64

```

You can also think of a series as a fixed-length *ordered dict*, since it is an assignment of index values to data values. It can be used in many contexts where you could use a *dict*:

```
[12]: "2022-02-02" in s2
```

```
[12]: True
```

```
[13]: "2022-02-09" in s2
```

```
[13]: False
```

Missing data

I will use NA and null synonymously to indicate missing data. The functions `isna` and `notna` in pandas should be used to identify missing data:

```
[14]: pd.isna(s2)
```

```
[14]: 2022-01-31    False
      2022-02-01    False
      2022-02-02    False
      2022-02-03    False
      2022-02-04    False
      2022-02-05    False
      2022-02-06    False
      Freq: D, dtype: bool
```

```
[15]: pd.notna(s2)
```

```
[15]: 2022-01-31     True
      2022-02-01     True
      2022-02-02     True
      2022-02-03     True
      2022-02-04     True
      2022-02-05     True
      2022-02-06     True
      Freq: D, dtype: bool
```

Series also has these as instance methods:

```
[16]: s2.isna()
```

```
[16]: 2022-01-31    False
      2022-02-01    False
      2022-02-02    False
      2022-02-03    False
      2022-02-04    False
      2022-02-05    False
      2022-02-06    False
      Freq: D, dtype: bool
```

Dealing with missing data is discussed in more detail in the section [Managing missing data with pandas](#).

A useful feature of Series for many applications is the automatic alignment by index labels in arithmetic operations:

```
[17]: idx = pd.date_range("2022-02-01", periods=7)

s3 = pd.Series(rng.normal(size=7), index=idx)
```

```
[18]: s2, s3
```

```
[18]: (2022-01-31    0.434474
      2022-02-01   -1.696645
      2022-02-02   -1.180240
      2022-02-03   -0.205702
      2022-02-04   -0.426140
      2022-02-05   -0.123695
      2022-02-06    1.071786
      Freq: D, dtype: float64,
      2022-02-01   -0.105019
      2022-02-02    0.156524
      2022-02-03    0.191187
      2022-02-04    0.002915
      2022-02-05    0.274354
      2022-02-06   -0.991969
      2022-02-07   -0.087003
      Freq: D, dtype: float64)
```

```
[19]: s2 + s3
```

```
[19]: 2022-01-31         NaN
      2022-02-01   -1.801664
      2022-02-02   -1.023716
      2022-02-03   -0.014515
      2022-02-04   -0.423225
      2022-02-05    0.150659
      2022-02-06    0.079817
      2022-02-07         NaN
      Freq: D, dtype: float64
```

If you have experience with SQL, this is similar to a [JOIN](#) operation.

Both the Series object itself and its index have a `name` attribute that can be integrated into other areas of the pandas functionality:

```
[20]: s3.name = "floats"
      s3.index.name = "date"
```

```
s3
```

```
[20]: date
      2022-02-01   -0.105019
      2022-02-02    0.156524
      2022-02-03    0.191187
      2022-02-04    0.002915
      2022-02-05    0.274354
      2022-02-06   -0.991969
      2022-02-07   -0.087003
      Freq: D, Name: floats, dtype: float64
```

DataFrame

A DataFrame represents a rectangular data table and contains an ordered, named collection of columns, each of which can have a different value type. The DataFrame has both a row index and a column index.

Note:

Although a DataFrame is two-dimensional, you can also use it to represent higher-dimensional data in a table format with hierarchical indexing using `join`, `combine` and `Reshaping`.

```
[21]: data = {
    "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
    "Decimal": [0, 1, 2, 3, 4, 5],
    "Octal": ["001", "002", "003", "004", "004", "005"],
    "Key": ["NUL", "Ctrl-A", "Ctrl-B", "Ctrl-C", "Ctrl-D", "Ctrl-E"],
}

df = pd.DataFrame(data)

df
```

```
[21]:
```

	Code	Decimal	Octal	Key
0	U+0000	0	001	NUL
1	U+0001	1	002	Ctrl-A
2	U+0002	2	003	Ctrl-B
3	U+0003	3	004	Ctrl-C
4	U+0004	4	004	Ctrl-D
5	U+0005	5	005	Ctrl-E

For large DataFrames, the `head` method selects only the first five rows:

```
[22]: df.head()

[22]:
```

	Code	Decimal	Octal	Key
0	U+0000	0	001	NUL
1	U+0001	1	002	Ctrl-A
2	U+0002	2	003	Ctrl-B
3	U+0003	3	004	Ctrl-C
4	U+0004	4	004	Ctrl-D

You can also specify columns and their order:

```
[23]: pd.DataFrame(data, columns=["Code", "Key"])

[23]:
```

	Code	Key
0	U+0000	NUL
1	U+0001	Ctrl-A
2	U+0002	Ctrl-B
3	U+0003	Ctrl-C
4	U+0004	Ctrl-D
5	U+0005	Ctrl-E

If you want to pass a column that is not contained in the dict, it will appear without values in the result:

```
[24]: df2 = pd.DataFrame(
    data, columns=["Code", "Decimal", "Octal", "Description", "Key"]
```

(continues on next page)

(continued from previous page)

)

df2

```
[24]:
```

	Code	Decimal	Octal	Description	Key
0	U+0000	0	001	NaN	NUL
1	U+0001	1	002	NaN	Ctrl-A
2	U+0002	2	003	NaN	Ctrl-B
3	U+0003	3	004	NaN	Ctrl-C
4	U+0004	4	004	NaN	Ctrl-D
5	U+0005	5	005	NaN	Ctrl-E

You can retrieve a column in a DataFrame with a dict-like notation:

```
[25]: df["Code"]
```

```
[25]:
```

0	U+0000
1	U+0001
2	U+0002
3	U+0003
4	U+0004
5	U+0005

Name: Code, dtype: object

This way you can also make a column the index:

```
[26]: df2 = pd.DataFrame(
    data, columns=["Decimal", "Octal", "Description", "Key"], index=df["Code"]
)
```

df2

```
[26]:
```

	Decimal	Octal	Description	Key
Code				
U+0000	0	001	NaN	NUL
U+0001	1	002	NaN	Ctrl-A
U+0002	2	003	NaN	Ctrl-B
U+0003	3	004	NaN	Ctrl-C
U+0004	4	004	NaN	Ctrl-D
U+0005	5	005	NaN	Ctrl-E

Rows can be retrieved by position or name with the `pandas.DataFrame.loc` attribute:

```
[27]: df2.loc["U+0001"]
```

```
[27]:
```

Decimal	1
Octal	002
Description	NaN
Key	Ctrl-A

Name: U+0001, dtype: object

Column values can be changed by assignment. For example, a scalar value or an array of values could be assigned to the empty *Description* column:

```
[28]: df2["Description"] = [
    "Null character",
    "Start of Heading",
    "Start of Text",
    "End-of-text character",
    "End-of-transmission character",
    "Enquiry character",
]

df2
```

```
[28]:
```

	Decimal	Octal	Description	Key
Code				
U+0000	0	001	Null character	NUL
U+0001	1	002	Start of Heading	Ctrl-A
U+0002	2	003	Start of Text	Ctrl-B
U+0003	3	004	End-of-text character	Ctrl-C
U+0004	4	004	End-of-transmission character	Ctrl-D
U+0005	5	005	Enquiry character	Ctrl-E

Assigning a non-existing column creates a new column.

Columns can be removed with `pandas.DataFrame.drop` and displayed with `pandas.DataFrame.columns`:

```
[29]: df3 = df2.drop(columns=["Decimal", "Octal"])
```

```
[30]: df2.columns
```

```
[30]: Index(['Decimal', 'Octal', 'Description', 'Key'], dtype='object')
```

```
[31]: df3.columns
```

```
[31]: Index(['Description', 'Key'], dtype='object')
```

Another common form of data is nested dict of dicts:

```
[32]: u = {
    "U+0006": {
        "Decimal": "6",
        "Octal": "006",
        "Description": "Acknowledge character",
        "Key": "Ctrl-F",
    },
    "U+0007": {
        "Decimal": "7",
        "Octal": "007",
        "Description": "Bell character",
        "Key": "Ctrl-G",
    },
}

df4 = pd.DataFrame(u)

df4
```

```
[32]:
```

		U+0006	U+0007
Decimal		6	7
Octal		006	007
Description	Acknowledge character	Bell character	
Key	Ctrl-F	Ctrl-G	

You can transpose the DataFrame, i.e. swap the rows and columns, with a similar syntax to a NumPy array:

```
[33]: df4.T
```

```
[33]:
```

	Decimal	Octal		Description	Key
U+0006	6	006		Acknowledge character	Ctrl-F
U+0007	7	007		Bell character	Ctrl-G

Warning:

Note that when transposing, the data types of the columns are discarded if the columns do not all have the same data type, so when transposing and then transposing back, the previous type information may be lost. In this case, the columns become arrays of pure Python objects.

The keys in the inner dicts are combined to form the index in the result. This is not the case when an explicit index is specified:

```
[34]: df5 = pd.DataFrame(u, index=["Decimal", "Octal", "Key"])
df5
```

```
[34]:
```

	U+0006	U+0007
Decimal	6	7
Octal	006	007
Key	Ctrl-F	Ctrl-G

2.4.2 Converting Python data structures into pandas

Python data structures such as lists and arrays can be converted into pandas *Series* or *DataFrames*.

```
[1]: import numpy as np
import pandas as pd
```

Series

Python *lists* can easily be converted into pandas *Series*:

```
[2]: list1 = [-0.751442, 0.816935, -0.272546, -0.268295, -0.296728, 0.176255, -0.322612]
pd.Series(list1)
```

```
[2]: 0    -0.751442
1     0.816935
2    -0.272546
3    -0.268295
4    -0.296728
5     0.176255
6    -0.322612
dtype: float64
```

Multiple lists can also be easily converted into one pandas Series:

```
[3]: list2 = [-0.029608, -0.277982, 2.693057, -0.850817, 0.783868, -1.137835, -0.617132]

pd.Series(list1 + list2)

[3]: 0    -0.751442
     1     0.816935
     2    -0.272546
     3    -0.268295
     4    -0.296728
     5     0.176255
     6    -0.322612
     7    -0.029608
     8    -0.277982
     9     2.693057
    10    -0.850817
    11     0.783868
    12    -1.137835
    13    -0.617132
dtype: float64
```

A list can also be passed as an index:

```
[4]: date = [
      "2022-01-31",
      "2022-02-01",
      "2022-02-02",
      "2022-02-03",
      "2022-02-04",
      "2022-02-05",
      "2022-02-06",
      ]

pd.Series(list1, index=date)

[4]: 2022-01-31    -0.751442
     2022-02-01     0.816935
     2022-02-02    -0.272546
     2022-02-03    -0.268295
     2022-02-04    -0.296728
     2022-02-05     0.176255
     2022-02-06    -0.322612
dtype: float64
```

With Python [dictionaries](#) you can pass not only values but also the corresponding keys to a pandas series:

```
[5]: dict1 = {
      "2022-01-31": -0.751442,
      "2022-02-01": 0.816935,
      "2022-02-02": -0.272546,
      "2022-02-03": -0.268295,
      "2022-02-04": -0.296728,
      "2022-02-05": 0.176255,
      "2022-02-06": -0.322612,
```

(continues on next page)

(continued from previous page)

```
}
pd.Series(dict1)
```

```
[5]: 2022-01-31    -0.751442
      2022-02-01     0.816935
      2022-02-02    -0.272546
      2022-02-03    -0.268295
      2022-02-04    -0.296728
      2022-02-05     0.176255
      2022-02-06    -0.322612
      dtype: float64
```

When you pass a dict, the index in the resulting pandas series takes into account the order of the keys in the dict.

With `collections.ChainMap` you can also turn several dicts into one pandas.Series.

First we define a second dict:

```
[6]: dict2 = {
      "2022-02-07": -0.029608,
      "2022-02-08": -0.277982,
      "2022-02-09": 2.693057,
      "2022-02-10": -0.850817,
      "2022-02-11": 0.783868,
      "2022-02-12": -1.137835,
      "2022-02-13": -0.617132,
      }
```

```
[7]: from collections import ChainMap
```

```
pd.Series(ChainMap(dict1, dict2))
```

```
[7]: 2022-02-07    -0.029608
      2022-02-08    -0.277982
      2022-02-09     2.693057
      2022-02-10    -0.850817
      2022-02-11     0.783868
      2022-02-12    -1.137835
      2022-02-13    -0.617132
      2022-01-31    -0.751442
      2022-02-01     0.816935
      2022-02-02    -0.272546
      2022-02-03    -0.268295
      2022-02-04    -0.296728
      2022-02-05     0.176255
      2022-02-06    -0.322612
      dtype: float64
```

DataFrame

Lists of lists can be loaded into a pandas DataFrame with:

```
[8]: df = pd.DataFrame([list1, list2])
df
```

	0	1	2	3	4	5	6
0	-0.751442	0.816935	-0.272546	-0.268295	-0.296728	0.176255	-0.322612
1	-0.029608	-0.277982	2.693057	-0.850817	0.783868	-1.137835	-0.617132

You can also transfer a list into a DataFrame index:

```
[9]: pd.DataFrame([list1, list2], index=["2022-01-31", "2022-02-01"])
[9]:
```

	0	1	2	3	4	5
2022-01-31	-0.751442	0.816935	-0.272546	-0.268295	-0.296728	0.176255
2022-02-01	-0.029608	-0.277982	2.693057	-0.850817	0.783868	-1.137835

```

      6
2022-01-31 -0.322612
2022-02-01 -0.617132

```

A pandas DataFrame can be created from a dict with values in lists:

```
[10]: data = {
    "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
    "Decimal": [0, 1, 2, 3, 4, 5],
    "Octal": ["001", "002", "003", "004", "004", "005"],
    "Key": ["NUL", "Ctrl-A", "Ctrl-B", "Ctrl-C", "Ctrl-D", "Ctrl-E"],
}
```

```
[11]: pd.DataFrame(data)
```

	Code	Decimal	Octal	Key
0	U+0000	0	001	NUL
1	U+0001	1	002	Ctrl-A
2	U+0002	2	003	Ctrl-B
3	U+0003	3	004	Ctrl-C
4	U+0004	4	004	Ctrl-D
5	U+0005	5	005	Ctrl-E

Another common form of data is nested dict of dicts:

```
[12]: data2 = {
        "U+0006": {"Decimal": "6", "Octal": "006", "Key": "Ctrl-F"},
        "U+0007": {"Decimal": "7", "Octal": "007", "Key": "Ctrl-G"},
    }

    df2 = pd.DataFrame(data2)

    df2
```

```
[12]:
```

	U+0006	U+0007
Decimal	6	7

(continues on next page)

(continued from previous page)

Octal	006	007
Key	Ctrl-F	Ctrl-G

Dicts of Series are treated in a similar way:

```
[13]: data3 = {"U+0006": df2["U+0006"][2:], "U+0007": df2["U+0007"][2:]}
      pd.DataFrame(data3)
[13]:      U+0006  U+0007
      Key  Ctrl-F  Ctrl-G
```

2.4.3 Indexing

Index objects

The index objects of pandas are responsible for the axis labels and other metadata, such as the axis name. Any array or other sequence of labels you use when constructing a series or DataFrame is internally converted into an index:

```
[1]: import pandas as pd

obj = pd.Series(range(7), index=pd.date_range("2022-02-02", periods=7))

[2]: obj.index
[2]: DatetimeIndex(['2022-02-02', '2022-02-03', '2022-02-04', '2022-02-05',
                  '2022-02-06', '2022-02-07', '2022-02-08'],
                  dtype='datetime64[ns]', freq='D')

[3]: obj.index[3:]
[3]: DatetimeIndex(['2022-02-05', '2022-02-06', '2022-02-07', '2022-02-08'], dtype=
      ↪ 'datetime64[ns]', freq='D')
```

Index objects are immutable and therefore cannot be changed by the user:

```
[4]: obj.index[1] = "2022-02-03"

-----
TypeError                                Traceback (most recent call last)
Cell In[4], line 1
----> 1 obj.index[1] = "2022-02-03"

File ~/local/share/venv/pythons311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
  ↪ core/indexes/base.py:5157, in Index.__setitem__(self, key, value)
    5155 @final
    5156 def __setitem__(self, key, value):
-> 5157     raise TypeError("Index does not support mutable operations")

TypeError: Index does not support mutable operations
```

Immutability makes the sharing of index objects in data structures more secure:

```
[5]: import numpy as np
```

```
labels = pd.Index(np.arange(3))
```

```
labels
```

```
[5]: Index([0, 1, 2], dtype='int64')
```

```
[6]: rng = np.random.default_rng()
obj2 = pd.Series(rng.normal(size=3), index=labels)
```

```
[7]: obj2
```

```
[7]: 0    0.515353
     1    1.153708
     2   -1.776476
     dtype: float64
```

```
[8]: obj2.index is labels
```

```
[8]: True
```

To be similar to an array, an index also behaves like a fixed-size set:

```
[9]: data1 = {
      "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
      "Decimal": [0, 1, 2, 3, 4, 5],
      "Octal": ["001", "002", "003", "004", "004", "005"],
    }
df1 = pd.DataFrame(data1)
```

```
[10]: df1
```

```
[10]:   Code  Decimal  Octal
0  U+0000         0   001
1  U+0001         1   002
2  U+0002         2   003
3  U+0003         3   004
4  U+0004         4   004
5  U+0005         5   005
```

```
[11]: df1.columns
```

```
[11]: Index(['Code', 'Decimal', 'Octal'], dtype='object')
```

```
[12]: "Code" in df1.columns
```

```
[12]: True
```

```
[13]: "Key" in df1.columns
```

```
[13]: False
```


Axis indices with double labels

Unlike Python sets, a pandas index can contain duplicate labels:

```
[14]: data2 = {
      "Code": ["U+0006", "U+0007"],
      "Decimal": [6, 7],
      "Octal": ["006", "007"],
    }
df2 = pd.DataFrame(data2)
df12 = pd.concat([df1, df2])

df12
```

```
[14]:
```

	Code	Decimal	Octal
0	U+0000	0	001
1	U+0001	1	002
2	U+0002	2	003
3	U+0003	3	004
4	U+0004	4	004
5	U+0005	5	005
0	U+0006	6	006
1	U+0007	7	007

When *selecting* duplicate labels, all occurrences of the label in question are selected:

```
[15]: df12.loc[1]
```

```
[15]:
```

	Code	Decimal	Octal
1	U+0001	1	002
1	U+0007	7	007

```
[16]: df12.loc[2]
```

```
[16]: Code      U+0002
      Decimal      2
      Octal      003
      Name: 2, dtype: object
```

Data selection is one of the main points that behaves differently with duplicates. Indexing a label with multiple entries results in a series, while single entries result in a scalar value. This can complicate your code because the output type of indexing can vary depending on whether a label is repeated or not. In addition, many pandas functions, such as `reindex`, require labels to be unique. You can use the `is_unique` property of the index to determine whether its labels are unique or not:

```
[17]: df12.index.is_unique
```

```
[17]: False
```

To avoid duplicate labels, you can use `ignore_index=True`, for example:

```
[18]: df12 = pd.concat([df1, df2], ignore_index=True)
```

```
df12
```

```
[18]:
```

	Code	Decimal	Octal
0	U+0000	0	001
1	U+0001	1	002
2	U+0002	2	003
3	U+0003	3	004
4	U+0004	4	004
5	U+0005	5	005
6	U+0006	6	006
7	U+0007	7	007

Some index methods and properties

Each index has a number of set logic methods and properties that answer other general questions about the data it contains. The following are some useful methods and properties:

Method	Description
<code>concat</code>	concatenates additional index objects, creating a new index
<code>Index.difference</code>	calculates the difference of two sets as an index
<code>Index.intersection</code>	calculates the intersection
<code>Index.union</code>	calculates the union set
<code>Index.isin</code>	computes a boolean array indicating whether each value is contained in the passed collection
<code>Index.delete</code>	computes a new index by deleting the element in index <code>i</code>
<code>Index.drop</code>	computes a new index by deleting the passed values
<code>Index.insert</code>	insert computes new index by inserting the element in index <code>i</code>
<code>Index.is_monotonic_increasing</code>	<code>is_monotonic</code> returns True if each element is greater than or equal to the previous element
<code>Index.is_unique</code>	<code>is_unique</code> returns True if the index does not contain duplicate values
<code>Index.unique</code>	calculates the array of unique values in the index

Re-indexing with `Index.reindex`

An important method for Pandas objects is `Index.reindex`, which can be used to create a new object with rearranged values that match the new index. Consider, for example:

```
[19]: obj = pd.Series(range(7), index=pd.date_range("2022-02-02", periods=7))
```

```
[20]: obj
```

```
[20]:
```

2022-02-02	0
2022-02-03	1
2022-02-04	2
2022-02-05	3
2022-02-06	4
2022-02-07	5
2022-02-08	6

Freq: D, dtype: int64

```
[21]: new_index = pd.date_range("2022-02-03", periods=7)
```

```
[22]: obj.reindex(new_index)
```

```
[22]: 2022-02-03    1.0
      2022-02-04    2.0
      2022-02-05    3.0
      2022-02-06    4.0
      2022-02-07    5.0
      2022-02-08    6.0
      2022-02-09    NaN
      Freq: D, dtype: float64
```

`Index.reindex` creates a new index and re-indexes the DataFrame. By default, values in the new index for which there are no corresponding records in the DataFrame become NaN.

For ordered data such as time series, it may be desirable to interpolate or fill values during reindexing. The `method` option allows this with a method like `ffill` that fills the values forward:

```
[23]: obj.reindex(new_index, method="ffill")
```

```
[23]: 2022-02-03    1
      2022-02-04    2
      2022-02-05    3
      2022-02-06    4
      2022-02-07    5
      2022-02-08    6
      2022-02-09    6
      Freq: D, dtype: int64
```

For a DataFrame, `reindex` can change either the (row) index, the columns or both. If only a sequence is passed, the rows in the result are re-indexed:

```
[24]: df1.reindex(range(7))
```

```
[24]:      Code  Decimal  Octal
0  U+0000      0.0    001
1  U+0001      1.0    002
2  U+0002      2.0    003
3  U+0003      3.0    004
4  U+0004      4.0    004
5  U+0005      5.0    005
6      NaN      NaN    NaN
```

The columns can be re-indexed with the keyword `columns`:

```
[25]: encoding = ["Octal", "Code", "Description"]
```

```
df1.reindex(columns=encoding)
```

```
[25]:   Octal   Code  Description
0    001  U+0000           NaN
1    002  U+0001           NaN
2    003  U+0002           NaN
3    004  U+0003           NaN
4    004  U+0004           NaN
5    005  U+0005           NaN
```

Arguments of the function `Index.reindex`

Argument	Description
<code>labels</code>	New sequence to be used as index. Can be an index instance or another sequence-like Python data structure. An index is used exactly as it is, without being copied.
<code>axis</code>	The new axis to index, either <code>index</code> (rows) or <code>columns</code> . The default is <code>index</code> . You can alternatively use <code>reindex(index=new_labels)</code> or <code>reindex(columns=new_labels)</code> .
<code>method</code>	Interpolation method; <code>ffill</code> fills forwards, while <code>bfill</code> fills backwards.
<code>fill_value</code>	Substitute value to be used when missing data is inserted by re-indexing. Uses <code>fill_value='missing'</code> (the default behaviour) if the missing labels in the result are to have zero values.
<code>limit</code>	When filling forward or backward, the maximum number of elements to fill.
<code>tolerance</code>	When filling forward or backward, the maximum size of the gap to be filled for inexact matches.
<code>level</code>	Match single index at <code>MultiIndex</code> level; otherwise select subset.
<code>copy</code>	If <code>True</code> , the underlying data is always copied, even if the new index matches the old index; if <code>False</code> , the data is not copied if the indices are equivalent.

Rename axis indices

The axis labels can be converted by a function or mapping to create new, differently labelled objects. You can also change the axes in place without creating a new data structure. Here is a simple example:

```
[26]: df3 = pd.DataFrame(
        np.arange(12).reshape((3, 4)),
        index=["Deutsch", "English", "Français"],
        columns=[1, 2, 3, 4],
    )

df3
```

```
[26]:      1  2  3  4
Deutsch  0  1  2  3
English  4  5  6  7
Français 8  9 10 11
```

Rename axis indices with `Index.map`

The axis labels can be converted by a function or `Index.map` to create new, differently labeled objects. You can also change the axes in place without creating a new data structure. Here is a simple example:

```
[27]: transform = lambda x: x[:2].upper()

df3.index.map(transform)

[27]: Index(['DE', 'EN', 'FR'], dtype='object')
```

You can assign the index and change the DataFrame on the spot:

```
[28]: df3.index = df3.index.map(transform)

df3
```

```
[28]:      1  2   3   4
DE  0  1   2   3
EN  4  5   6   7
FR  8  9  10  11
```

Rename axis indices with `Index.rename`

If you want to create a converted version of your dataset without changing the original, you can use `Index.rename`:

```
[29]: df3.rename(index=str.lower)
```

```
[29]:      1  2   3   4
de  0  1   2   3
en  4  5   6   7
fr  8  9  10  11
```

In particular, `Index.rename` can be used in conjunction with a dict-like object that provides new values for a subset of the axis labels:

```
[30]: df3.rename(
        index={"DE": "BE", "EN": "DE", "FR": "EN"},
        columns={1: 0, 2: 1, 3: 2, 4: 3},
        inplace=True,
    )
```

df3

```
[30]:      0  1   2   3
BE  0  1   2   3
DE  4  5   6   7
EN  8  9  10  11
```

`Index.rename` saves you from manually copying the DataFrame and assigning its index and column attributes. If you want to change a data set on the spot, also pass `inplace=True`:

```
[31]: df3.rename(
        index={"DE": "BE", "EN": "DE", "FR": "EN"},
        columns={1: 0, 2: 1, 3: 2, 4: 3},
        inplace=True,
    )
```

df3

```
[31]:      0  0   1   2
BE  0  1   2   3
BE  4  5   6   7
DE  8  9  10  11
```

Hierarchical Indexing

Hierarchical indexing is an important feature of pandas that allows you to have multiple index levels on one axis. This gives you the opportunity to work with higher dimensional data in a lower dimensional form.

Let's start with a simple example: Let's create a series of lists as an index:

```
[32]: hits = pd.Series(
    [83080, 20336, 11376, 1228, 468],
    index=[
        [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            "Python Basics",
            "Python Basics",
        ],
        ["de", "en", "de", "de", "en"],
    ],
)

hits
```

```
[32]: Jupyter Tutorial  de    83080
      Jupyter Tutorial  en    20336
      PyViz Tutorial    de    11376
      Python Basics    de     1228
      Python Basics    en     468
dtype: int64
```

What you see is a graphical view of a series with a `pandas.MultiIndex`. The *gaps* in the index display mean that the label above it is to be used.

```
[33]: hits.index
[33]: MultiIndex([('Jupyter Tutorial', 'de'),
                  ('Jupyter Tutorial', 'en'),
                  ('PyViz Tutorial', 'de'),
                  ('Python Basics', 'de'),
                  ('Python Basics', 'en')],
                 )
```

With a hierarchically indexed object, so-called partial indexing is possible, with which you can select subsets of the data:

```
[34]: hits["Jupyter Tutorial"]
```

```
[34]: de    83080
      en    20336
dtype: int64
```

```
[35]: hits["Jupyter Tutorial":"Python Basics"]
```

```
[35]: Jupyter Tutorial  de    83080
      Jupyter Tutorial  en    20336
      PyViz Tutorial    de    11376
```

(continues on next page)

(continued from previous page)

```
Python Basics    de    1228
                  en    468
dtype: int64
```

```
[36]: hits.loc[["Jupyter Tutorial", "Python Basics"]]
```

```
[36]: Jupyter Tutorial    de    83080
                  en    20336
Python Basics          de    1228
                  en    468
dtype: int64
```

The selection is even possible from an *inner* level. In the following I select all values with the value 1 from the second index level:

```
[37]: hits.loc[:, "de"]
```

```
[37]: Jupyter Tutorial    83080
PyViz Tutorial          11376
Python Basics          1228
dtype: int64
```

View vs. copy

In Pandas, whether you get a view or not depends on the structure and data types of the original DataFrame – and whether changes made to a view are propagated back to the original DataFrame.

stack and unstack

Hierarchical indexing plays an important role in data reshaping and group-based operations such as forming a *pivot table*. For example, you can reorder this data into a DataFrame using the `pandas.Series.unstack` method:

```
[38]: hits.unstack()
```

```
[38]:              de      en
Jupyter Tutorial  83080.0  20336.0
PyViz Tutorial    11376.0     NaN
Python Basics    1228.0   468.0
```

The reverse operation of `unstack` is `stack`:

```
[39]: hits.unstack().stack()
```

```
[39]: Jupyter Tutorial    de    83080.0
                  en    20336.0
PyViz Tutorial          de    11376.0
Python Basics          de    1228.0
                  en    468.0
dtype: float64
```

In a DataFrame, each axis can have a hierarchical index:

```
[40]: version_hits = [
    [19651, 0, 30134, 0, 33295, 0],
    [4722, 1825, 3497, 2576, 4009, 3707],
    [2573, 0, 4873, 0, 3930, 0],
    [525, 0, 427, 0, 276, 0],
    [157, 0, 85, 0, 226, 0],
]

df = pd.DataFrame(
    version_hits,
    index=[
        [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            "Python Basics",
            "Python Basics",
        ],
        ["de", "en", "de", "de", "en"],
    ],
    columns=[
        ["12/2021", "12/2021", "01/2022", "01/2022", "02/2022", "02/2022"],
        ["latest", "stable", "latest", "stable", "latest", "stable"],
    ],
)

df
```

```
[40]:
```

		12/2021		01/2022		02/2022	
		latest	stable	latest	stable	latest	stable
Jupyter Tutorial	de	19651	0	30134	0	33295	0
	en	4722	1825	3497	2576	4009	3707
PyViz Tutorial	de	2573	0	4873	0	3930	0
Python Basics	de	525	0	427	0	276	0
	en	157	0	85	0	226	0

The hierarchy levels can have names (as strings or any Python objects). If this is the case, they are displayed in the console output:

```
[41]: df.index.names = ["Title", "Language"]
df.columns.names = ["Month", "Version"]

df
```

```
[41]:
```

		12/2021		01/2022		02/2022	
		latest	stable	latest	stable	latest	stable
Title	Language						
Jupyter Tutorial	de	19651	0	30134	0	33295	0
	en	4722	1825	3497	2576	4009	3707
PyViz Tutorial	de	2573	0	4873	0	3930	0
Python Basics	de	525	0	427	0	276	0
	en	157	0	85	0	226	0

Warning:

Make sure that the index names `Month` and `Version` are not part of the row names (of the `df.index` values).

With the partial column indexing you can select column groups in a similar way:

```
[42]: df["12/2021"]
```

```
[42]: Version          latest  stable
Title      Language
Jupyter Tutorial de      19651      0
           en      4722    1825
PyViz Tutorial  de      2573      0
Python Basics  de       525      0
           en       157      0
```

With `MultiIndex.from_arrays`, a `MultiIndex` can be created itself and then reused; the columns in the preceding `DataFrame` with level names could be created in this way:

```
[43]: pd.MultiIndex.from_arrays(
    [
        [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            "Python Basics",
            "Python Basics",
        ],
        ["de", "en", "de", "de", "en"],
    ],
    names=["Title", "Language"],
)
```

```
[43]: MultiIndex([('Jupyter Tutorial', 'de'),
                  ('Jupyter Tutorial', 'en'),
                  ('PyViz Tutorial', 'de'),
                  ('Python Basics', 'de'),
                  ('Python Basics', 'en')],
                  names=['Title', 'Language'])
```

Rearranging and Sorting Levels

There may be times when you want to rearrange the order of the levels on an axis or sort the data by the values in a particular level. The function `DataFrame.swaplevel` takes two level numbers or names and returns a new object in which the levels are swapped (but the data remains unchanged):

```
[44]: df.swaplevel("Language", "Title")
```

```
[44]: Month          12/2021      01/2022      02/2022
Version          latest stable latest stable latest stable
Language Title
de      Jupyter Tutorial  19651      0    30134      0    33295      0
en      Jupyter Tutorial  4722    1825    3497    2576    4009    3707
de      PyViz Tutorial    2573      0    4873      0    3930      0
           Python Basics    525      0    427      0    276      0
en      Python Basics    157      0     85      0    226      0
```

`DataFrame.sort_index`, on the other hand, sorts the data only by the values in a single level. When swapping levels, it is not uncommon to also use `sort_index` so that the result is lexicographically sorted by the specified level:

```
[45]: df.sort_index(level=0)
```

```
[45]: Month          12/2021          01/2022          02/2022
      Version          latest stable  latest stable  latest stable
      Title    Language
Jupyter Tutorial de      19651         0    30134         0    33295         0
              en      4722      1825    3497      2576    4009      3707
PyViz Tutorial  de      2573         0    4873         0    3930         0
Python Basics  de       525         0     427         0     276         0
              en       157         0      85         0     226         0
```

However, the *PyViz Tutorial* will now be sorted before the *Python Basics*, as all upper case letters appear before lower case letters in this sorting. To avoid this, you can use the following lambda function:

```
[46]: df.sort_index(level=0, key=lambda x: x.str.lower())
```

```
[46]: Month          12/2021          01/2022          02/2022
      Version          latest stable  latest stable  latest stable
      Title    Language
Jupyter Tutorial de      19651         0    30134         0    33295         0
              en      4722      1825    3497      2576    4009      3707
Python Basics  de       525         0     427         0     276         0
              en       157         0      85         0     226         0
PyViz Tutorial  de      2573         0    4873         0    3930         0
```

```
[47]: df.swaplevel(0, 1).sort_index(level=0)
```

```
[47]: Month          12/2021          01/2022          02/2022
      Version          latest stable  latest stable  latest stable
      Language Title
de      Jupyter Tutorial  19651         0    30134         0    33295         0
      PyViz Tutorial    2573         0    4873         0    3930         0
      Python Basics     525         0     427         0     276         0
en      Jupyter Tutorial  4722      1825    3497      2576    4009      3707
      Python Basics     157         0      85         0     226         0
```

Note:

Data selection performance is much better for hierarchically indexed objects if the index is sorted lexicographically, starting with the outermost level, i.e. the result of calling `sort_index(level=0)` or `sort_index()`.

Summary statistics by level

Many descriptive and summary statistics for `DataFrame` and `Series` have a `level` option that allows you to specify the level by which you can aggregate on a particular axis. Consider the `DataFrame` above; we can aggregate either the rows or the columns by level as follows:

```
[48]: df.groupby(level="Language").sum()
```

```
[48]: Month    12/2021    01/2022    02/2022
      Version  latest stable  latest stable  latest stable
      Language
de      Jupyter Tutorial  19651         0    30134         0    33295         0
      PyViz Tutorial    2573         0    4873         0    3930         0
      Python Basics     525         0     427         0     276         0
en      Jupyter Tutorial  4722      1825    3497      2576    4009      3707
      Python Basics     157         0      85         0     226         0
```

(continues on next page)

(continued from previous page)

de	22749	0	35434	0	37501	0
en	4879	1825	3582	2576	4235	3707

```
[49]: df.groupby(level="Month", axis=1).sum()
```

```
[49]: Month          01/2022  02/2022  12/2021
Title      Language
Jupyter Tutorial de      30134    33295    19651
              en       6073     7716     6547
PyViz Tutorial  de       4873     3930     2573
Python Basics  de        427      276      525
              en         85      226      157
```

Internally, pandas' `DataFrame.groupby` machinery is used for this purpose, which is explained in more detail in [Group Operations](#).

Indexing with the columns of a DataFrame

It is not uncommon to use one or more columns of a DataFrame as a row index; alternatively, you can move the row index into the columns of the DataFrame. Here is an example DataFrame:

```
[50]: data = [
    ["Jupyter Tutorial", "de", 19651, 0, 30134, 0, 33295, 0],
    ["Jupyter Tutorial", "en", 4722, 1825, 3497, 2576, 4009, 3707],
    ["PyViz Tutorial", "de", 2573, 0, 4873, 0, 3930, 0],
    ["Python Basics", "de", 525, 0, 427, 0, 276, 0],
    ["Python Basics", "en", 157, 0, 85, 0, 226, 0],
]

df = pd.DataFrame(data)

df
```

```
[50]:
```

		0	1	2	3	4	5	6	7
0	Jupyter Tutorial	de	19651	0	30134	0	33295	0	
1	Jupyter Tutorial	en	4722	1825	3497	2576	4009	3707	
2	PyViz Tutorial	de	2573	0	4873	0	3930	0	
3	Python Basics	de	525	0	427	0	276	0	
4	Python Basics	en	157	0	85	0	226	0	

The function `pandas.DataFrame.set_index` creates a new DataFrame that uses one or more of its columns as an index:

```
[51]: df2 = df.set_index([0, 1])
```

```
df2
```

```
[51]:
```

		2	3	4	5	6	7
0	1						
Jupyter Tutorial	de	19651	0	30134	0	33295	0
	en	4722	1825	3497	2576	4009	3707
PyViz Tutorial	de	2573	0	4873	0	3930	0
Python Basics	de	525	0	427	0	276	0
	en	157	0	85	0	226	0

By default, the columns are removed from the DataFrame, but you can also leave them in by passing `drop=False` to `set_index`:

```
[52]: df.set_index([0, 1], drop=False)
```

```
[52]:
```

			0	1	2	3	4	5	6	\
0	1									
Jupyter Tutorial	de	Jupyter Tutorial	de	19651	0	30134	0	33295		
	en	Jupyter Tutorial	en	4722	1825	3497	2576	4009		
PyViz Tutorial	de	PyViz Tutorial	de	2573	0	4873	0	3930		
Python Basics	de	Python Basics	de	525	0	427	0	276		
	en	Python Basics	en	157	0	85	0	226		


```

0      1      7
Jupyter Tutorial de  0
                  en 3707
PyViz Tutorial   de  0
Python Basics   de  0
                  en  0

```

`DataFrame.reset_index`, on the other hand, does the opposite of `set_index`; the hierarchical index levels are moved into the columns:

```
[53]: df2.reset_index()
```

```
[53]:
```

		0	1	2	3	4	5	6	7
0	Jupyter Tutorial	de	19651	0	30134	0	33295	0	
1	Jupyter Tutorial	en	4722	1825	3497	2576	4009	3707	
2	PyViz Tutorial	de	2573	0	4873	0	3930	0	
3	Python Basics	de	525	0	427	0	276	0	
4	Python Basics	en	157	0	85	0	226	0	

2.4.4 Date and Time

With pandas you can create `Series` with date and time information. In the following we will show common operations with date data.

Note:

pandas supports dates stored in `UTC` values using the `datetime64[ns]` datatype. Local times from a single time zone are also supported. Multiple time zones are supported by a `pandas.Timestamp` object. If you need to handle times from multiple time zones, I would probably split the data by time zone and use a separate `DataFrame` or `Series` for each time zone.

See also:

- [Time series / date functionality](#)

Loading UTC time data

```
[1]: import pandas as pd

dt = pd.date_range("2022-03-27", periods=6, freq="H")

dt
[1]: DatetimeIndex(['2022-03-27 00:00:00', '2022-03-27 01:00:00',
                    '2022-03-27 02:00:00', '2022-03-27 03:00:00',
                    '2022-03-27 04:00:00', '2022-03-27 05:00:00'],
                    dtype='datetime64[ns]', freq='H')

[2]: utc = pd.to_datetime(dt, utc=True)

utc
[2]: DatetimeIndex(['2022-03-27 00:00:00+00:00', '2022-03-27 01:00:00+00:00',
                    '2022-03-27 02:00:00+00:00', '2022-03-27 03:00:00+00:00',
                    '2022-03-27 04:00:00+00:00', '2022-03-27 05:00:00+00:00'],
                    dtype='datetime64[ns, UTC]', freq='H')
```

Note:

The type of the result `dtype='datetime64[ns, UTC]'` indicates that the data is stored as UTC.

Let's convert this series to the time zone Europe/Berlin:

```
[3]: utc.tz_convert("Europe/Berlin")

[3]: DatetimeIndex(['2022-03-27 01:00:00+01:00', '2022-03-27 03:00:00+02:00',
                    '2022-03-27 04:00:00+02:00', '2022-03-27 05:00:00+02:00',
                    '2022-03-27 06:00:00+02:00', '2022-03-27 07:00:00+02:00'],
                    dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

Conversion of local time to UTC

```
[4]: local = utc.tz_convert("Europe/Berlin")

local.tz_convert("UTC")
[4]: DatetimeIndex(['2022-03-27 00:00:00+00:00', '2022-03-27 01:00:00+00:00',
                    '2022-03-27 02:00:00+00:00', '2022-03-27 03:00:00+00:00',
                    '2022-03-27 04:00:00+00:00', '2022-03-27 05:00:00+00:00'],
                    dtype='datetime64[ns, UTC]', freq='H')
```

Conversion to Unix time

If you have a `Series` with UTC or local time information, you can use this code to determine the seconds according to Unix time:

```
[5]: uts = pd.to_datetime(dt).view(int) / 10**9

uts

[5]: array([1.6483392e+09, 1.6483428e+09, 1.6483464e+09, 1.6483500e+09,
          1.6483536e+09, 1.6483572e+09])
```

To load the Unix time in UTC, you can proceed as follows:

```
[6]: (pd.to_datetime(uts, unit="s").tz_localize("UTC"))

[6]: DatetimeIndex(['2022-03-27 00:00:00+00:00', '2022-03-27 01:00:00+00:00',
                  '2022-03-27 02:00:00+00:00', '2022-03-27 03:00:00+00:00',
                  '2022-03-27 04:00:00+00:00', '2022-03-27 05:00:00+00:00'],
                  dtype='datetime64[ns, UTC]', freq=None)
```

Manipulation of dates

Convert to strings

With `pandas.DatetimeIndex` you have some possibilities to convert date and time into strings, for example into the name of the weekday:

```
[7]: local.day_name(locale="en_GB.UTF-8")

[7]: Index(['Sunday', 'Sunday', 'Sunday', 'Sunday', 'Sunday', 'Sunday'], dtype='object')
```

You can find out which locale is available to you with `locale -a`:

```
[8]: !locale -a

en_NZ
nl_NL.UTF-8
pt_BR.UTF-8
fr_CH.ISO8859-15
eu_ES.ISO8859-15
en_US.US-ASCII
af_ZA
bg_BG
cs_CZ.UTF-8
fi_FI
zh_CN.UTF-8
eu_ES
sk_SK.ISO8859-2
nl_BE
fr_BE
sk_SK
en_US.UTF-8
en_NZ.ISO8859-1
```

(continues on next page)

(continued from previous page)

```

de_CH
sk_SK.UTF-8
de_DE.UTF-8
am_ET.UTF-8
zh_HK
be_BY.UTF-8
uk_UA
pt_PT.ISO8859-1
en_AU.US-ASCII
kk_KZ.PT154
en_US
nl_BE.ISO8859-15
de_AT.ISO8859-1
hr_HR.ISO8859-2
fr_FR.ISO8859-1
af_ZA.UTF-8
am_ET
fi_FI.ISO8859-1
ro_RO.UTF-8
af_ZA.ISO8859-15
en_NZ.UTF-8
fi_FI.UTF-8
hr_HR.UTF-8
da_DK.UTF-8
ca_ES.ISO8859-1
en_AU.ISO8859-15
ro_RO.ISO8859-2
de_AT.UTF-8
pt_PT.ISO8859-15
sv_SE
fr_CA.ISO8859-1
fr_BE.ISO8859-1
en_US.ISO8859-15
it_CH.ISO8859-1
en_NZ.ISO8859-15
en_AU.UTF-8
de_AT.ISO8859-15
af_ZA.ISO8859-1
hu_HU.UTF-8
et_EE.UTF-8
he_IL.UTF-8
uk_UA.KOI8-U
be_BY
kk_KZ
hu_HU.ISO8859-2
it_CH
pt_BR
ko_KR
it_IT
fr_BE.UTF-8
ru_RU.ISO8859-5
zh_TW

```

(continues on next page)

(continued from previous page)

```
zh_CN.GB2312
no_NO.ISO8859-15
de_DE.ISO8859-15
en_CA
fr_CH.UTF-8
sl_SI.UTF-8
uk_UA.ISO8859-5
pt_PT
hr_HR
cs_CZ
fr_CH
he_IL
zh_CN.GBK
zh_CN.GB18030
fr_CA
pl_PL.UTF-8
ja_JP.SJIS
sr_YU.ISO8859-5
be_BY.CP1251
sr_YU.ISO8859-2
sv_SE.UTF-8
sr_YU.UTF-8
de_CH.UTF-8
sl_SI
pt_PT.UTF-8
ro_RO
en_NZ.US-ASCII
ja_JP
zh_CN
fr_CH.ISO8859-1
ko_KR.eucKR
be_BY.ISO8859-5
nl_NL.ISO8859-15
en_GB.ISO8859-1
en_CA.US-ASCII
is_IS.ISO8859-1
ru_RU.CP866
nl_NL
fr_CA.ISO8859-15
sv_SE.ISO8859-15
hy_AM
en_CA.ISO8859-15
en_US.ISO8859-1
zh_TW.Big5
ca_ES.UTF-8
ru_RU.CP1251
en_GB.UTF-8
en_GB.US-ASCII
ru_RU.UTF-8
eu_ES.UTF-8
es_ES.ISO8859-1
hu_HU
```

(continues on next page)

(continued from previous page)

```

el_GR.ISO8859-7
en_AU
it_CH.UTF-8
en_GB
sl_SI.ISO8859-2
ru_RU.KOI8-R
nl_BE.UTF-8
et_EE
fr_FR.ISO8859-15
cs_CZ.ISO8859-2
lt_LT.UTF-8
pl_PL.ISO8859-2
fr_BE.ISO8859-15
is_IS.UTF-8
tr_TR.ISO8859-9
da_DK.ISO8859-1
lt_LT.ISO8859-4
lt_LT.ISO8859-13
zh_TW.UTF-8
bg_BG.CP1251
el_GR.UTF-8
be_BY.CP1131
da_DK.ISO8859-15
is_IS.ISO8859-15
no_NO.ISO8859-1
nl_NL.ISO8859-1
nl_BE.ISO8859-1
sv_SE.ISO8859-1
pt_BR.ISO8859-1
zh_CN.eucCN
it_IT.UTF-8
en_CA.UTF-8
uk_UA.UTF-8
de_CH.ISO8859-15
de_DE.ISO8859-1
ca_ES
sr_YU
hy_AM.ARMSCII-8
ru_RU
zh_HK.UTF-8
eu_ES.ISO8859-1
is_IS
bg_BG.UTF-8
ja_JP.UTF-8
it_CH.ISO8859-15
fr_FR.UTF-8
ko_KR.UTF-8
et_EE.ISO8859-15
kk_KZ.UTF-8
ca_ES.ISO8859-15
en_IE.UTF-8
es_ES

```

(continues on next page)

(continued from previous page)

```
de_CH.ISO8859-1
en_CA.ISO8859-1
es_ES.ISO8859-15
en_AU.ISO8859-1
el_GR
da_DK
no_NO
it_IT.ISO8859-1
en_IE
zh_HK.Big5HKSCS
hi_IN.ISCII-DEV
ja_JP.eucJP
it_IT.ISO8859-15
pl_PL
ko_KR.CP949
fr_CA.UTF-8
fi_FI.ISO8859-15
en_GB.ISO8859-15
fr_FR
hy_AM.UTF-8
no_NO.UTF-8
es_ES.UTF-8
de_AT
tr_TR.UTF-8
de_DE
lt_LT
tr_TR
C
POSIX
```

Other attributes of `DatetimeIndex` that can be used to convert date and time into strings are:

Attribute	Description
year	the year as <code>datetime</code> .
month	the month as January 1 and December 12
day	the day of the <code>datetime</code>
hour	the hours of the <code>datetime</code>
minute	the minutes of the <code>datetime</code>
seconds	the seconds of the <code>datetime</code>
microsecond	the microseconds of the <code>datetime</code> .
nanosecond	the nanoseconds of <code>datetime</code>
date	returns a NumPy array of Python <code>datetime.date</code> objects
time	returns a NumPy array of <code>datetime.time</code> objects
timetz	returns a NumPy array of <code>datetime.time</code> objects with timezone information
dayofyear, day_of_year	the ordinal day of the year
dayofweek	the day of the week with Monday (0) and Sunday (6)
day_of_week	the day of the week with Monday (0) and Sunday (6)
weekday	the day of the week with Monday (0) and Sunday (6)
quarter	returns the quarter of the year
tz	returns the time zone
freq	returns the frequency object if it is set, otherwise <code>None</code>
freqstr	returns the frequency object as a string if it is set, otherwise <code>None</code>
is_month_start	indicates if the date is the first day of the month
is_month_end	indicates whether the date is the last day of the month
is_quarter_start	indicates whether the date is the first day of a quarter
is_quarter_end	shows if the date is the last day of a quarter
is_year_start	indicates whether the date is the first day of a year
is_year_end	indicates whether the date is the last day of a year
is_leap_year	Boolean indicator if the date falls in a leap year
inferred_freq	tries to return a string representing a frequency determined by <code>infer_freq</code>

However, there are also some methods with which you can convert the `DatetimeIndex` into strings, for example `strftime`:

```
[9]: local.strftime("%d.%m.%Y")
[9]: Index(['27.03.2022', '27.03.2022', '27.03.2022', '27.03.2022', '27.03.2022',
          '27.03.2022'],
          dtype='object')
```

Note:

In `strftime()` and `strptime()` [Format Codes](#) you get an overview of the different formatting possibilities of `strftime`.

Other methods are:

Method	Description
<code>normalize</code>	converts times to midnight
<code>strftime</code>	converts to index using the specified date format
<code>snap</code>	snaps the timestamp to the next occurring frequency
<code>tz_convert</code>	convert a tz capable datetime array/index from one time zone to another
<code>tz_localize</code>	localises tz-naive datetime array/index into tz-compatible datetime array/index
<code>round</code>	rounds the data up to the nearest specified frequency
<code>floor</code>	rounds the data down to the specified frequency
<code>ceil</code>	round the data to the specified frequency
<code>to_period</code>	converts the data to a PeriodArray/Index at a given frequency
<code>to_periodde</code>	calculates TimedeltaArray of the difference between the index values and the index converted to PeriodArray at the specified frequency
<code>to_pydateti</code>	returns Datetime array/index as ndarray object of <code>datetime.datetime</code> objects
<code>to_series</code>	creates a <code>Series</code> with index and values corresponding to index keys; useful with <code>map</code> for returning an indexer
<code>to_frame</code>	creates a <code>DataFrame</code> with a column containing the index
<code>month_name</code>	returns the month names of the <code>DatetimeIndex</code> with the specified locale
<code>day_name</code>	returns the day names of the <code>DatetimeIndex</code> with the specified locale
<code>mean</code>	returns the mean value of the array
<code>std</code>	returns the standard deviation of the sample across the requested axis

2.4.5 Select and filter data

Indexing series (`obj[...]`) works analogously to indexing NumPy arrays, except that you can use index values of the series instead of just integers. Here are some examples:

```
[1]: import numpy as np
import pandas as pd
```

```
[2]: idx = pd.date_range("2022-02-02", periods=7)
rng = np.random.default_rng()
s = pd.Series(rng.normal(size=7), index=idx)
```

```
[3]: s
[3]: 2022-02-02    0.002127
2022-02-03    1.655759
2022-02-04   -1.552128
2022-02-05   -1.581026
2022-02-06   -0.992316
2022-02-07    1.490786
2022-02-08   -1.542455
Freq: D, dtype: float64
```

```
[4]: s["2022-02-03"]
```

```
[4]: 1.655759430268265
```

```
[5]: s[1]
```

```
[5]: 1.655759430268265
```

```
[6]: s[2:4]
```

```
[6]: 2022-02-04    -1.552128
      2022-02-05    -1.581026
      Freq: D, dtype: float64
```

```
[7]: s[["2022-02-04", "2022-02-03", "2022-02-02"]]
```

```
[7]: 2022-02-04    -1.552128
      2022-02-03     1.655759
      2022-02-02     0.002127
      dtype: float64
```

```
[8]: s[[1, 3]]
```

```
[8]: 2022-02-03     1.655759
      2022-02-05    -1.581026
      Freq: 2D, dtype: float64
```

```
[9]: s[s > 0]
```

```
[9]: 2022-02-02     0.002127
      2022-02-03     1.655759
      2022-02-07     1.490786
      dtype: float64
```

While you can select data by label in this way, the preferred method for selecting index values is the `loc` operator:

```
[10]: s.loc[["2022-02-04", "2022-02-03", "2022-02-02"]]
```

```
[10]: 2022-02-04    -1.552128
      2022-02-03     1.655759
      2022-02-02     0.002127
      dtype: float64
```

The reason for the preference for `loc` is the different treatment of integers when indexing with `[]`. In regular `[]`-based indexing, integers are treated as labels if the index contains integers, so the behaviour varies depending on the data type of the index. In our example, the expression `s.loc[[3, 2, 1]]` will fail because the index does not contain integers:

```
[11]: s.loc[[3, 2, 1]]
```

```
-----
KeyError                                Traceback (most recent call last)
```

```
Cell In[11], line 1
```

```
----> 1 s.loc[[3, 2, 1]]
```

```
File ~/local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
  core/indexing.py:1103, in _LocationIndexer.__getitem__(self, key)
    1100 axis = self.axis or 0
    1102 maybe_callable = com.apply_if_callable(key, self.obj)
-> 1103 return self._getitem_axis(maybe_callable, axis=axis)
```

```
File ~/local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
```

(continues on next page)

(continued from previous page)

```

-> core/indexing.py:1332, in _LocIndexer._getitem_axis(self, key, axis)
    1329     if hasattr(key, "ndim") and key.ndim > 1:
    1330         raise ValueError("Cannot index with multidimensional key")
-> 1332     return self._getitem_iterable(key, axis=axis)
    1334 # nested tuple slicing
    1335 if is_nested_tuple(key, labels):

File ~/local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
-> core/indexing.py:1272, in _LocIndexer._getitem_iterable(self, key, axis)
    1269 self._validate_key(key, axis)
    1271 # A collection of keys
-> 1272 keyarr, indexer = self._get_listlike_indexer(key, axis)
    1273 return self.obj._reindex_with_indexers(
    1274     {axis: [keyarr, indexer]}, copy=True, allow_dups=True
    1275 )

File ~/local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
-> core/indexing.py:1462, in _LocIndexer._get_listlike_indexer(self, key, axis)
    1459 ax = self.obj._get_axis(axis)
    1460 axis_name = self.obj._get_axis_name(axis)
-> 1462 keyarr, indexer = ax._get_indexer_strict(key, axis_name)
    1464 return keyarr, indexer

File ~/local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
-> core/indexes/base.py:5877, in Index._get_indexer_strict(self, key, axis_name)
    5874 else:
    5875     keyarr, indexer, new_indexer = self._reindex_non_unique(keyarr)
-> 5877 self._raise_if_missing(keyarr, indexer, axis_name)
    5879 keyarr = self.take(indexer)
    5880 if isinstance(key, Index):
    5881     # GH 42790 - Preserve name from an Index

File ~/local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
-> core/indexes/base.py:5938, in Index._raise_if_missing(self, key, indexer, axis_name)
    5936     if use_interval_msg:
    5937         key = list(key)
-> 5938     raise KeyError(f"None of [{key}] are in the [{axis_name}]")
    5940 not_found = list(ensure_index(key)[missing_mask.nonzero()[0]].unique())
    5941 raise KeyError(f"{not_found} not in index")

KeyError: "None of [Index([3, 2, 1], dtype='int64')] are in the [index]"

```

While the `loc` operator exclusively indexes labels, the `iloc` operator exclusively indexes with integers:

```

[12]: s.iloc[[3, 2, 1]]
[12]: 2022-02-05    -1.581026
      2022-02-04    -1.552128
      2022-02-03     1.655759
      Freq: -1D, dtype: float64

```

You can also slice with labels, but this works differently from normal Python slicing because the endpoint is included:

```
[13]: s.loc["2022-02-03":"2022-02-04"]
```

```
[13]: 2022-02-03    1.655759
      2022-02-04   -1.552128
      Freq: D, dtype: float64
```

Setting with these methods changes the corresponding section of the row:

```
[14]: s.loc["2022-02-03":"2022-02-04"] = 0
```

s

```
[14]: 2022-02-02    0.002127
      2022-02-03    0.000000
      2022-02-04    0.000000
      2022-02-05   -1.581026
      2022-02-06   -0.992316
      2022-02-07    1.490786
      2022-02-08   -1.542455
      Freq: D, dtype: float64
```

Indexing in a DataFrame is used to retrieve one or more columns with either a single value or a sequence:

```
[15]: data = {
      "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
      "Decimal": [0, 1, 2, 3, 4, 5],
      "Octal": ["001", "002", "003", "004", "004", "005"],
      "Key": ["NUL", "Ctrl-A", "Ctrl-B", "Ctrl-C", "Ctrl-D", "Ctrl-E"],
      }

df = pd.DataFrame(data)
df = pd.DataFrame(data, columns=["Decimal", "Octal", "Key"], index=df["Code"])

df
```

```
[15]:
```

	Decimal	Octal	Key
Code			
U+0000	0	001	NUL
U+0001	1	002	Ctrl-A
U+0002	2	003	Ctrl-B
U+0003	3	004	Ctrl-C
U+0004	4	004	Ctrl-D
U+0005	5	005	Ctrl-E

```
[16]: df["Key"]
```

```
[16]: Code
      U+0000    NUL
      U+0001  Ctrl-A
      U+0002  Ctrl-B
      U+0003  Ctrl-C
      U+0004  Ctrl-D
      U+0005  Ctrl-E
      Name: Key, dtype: object
```

```
[17]: df[["Decimal", "Key"]]
```

```
[17]:      Decimal      Key
Code
U+0000      0      NUL
U+0001      1  Ctrl-A
U+0002      2  Ctrl-B
U+0003      3  Ctrl-C
U+0004      4  Ctrl-D
U+0005      5  Ctrl-E
```

```
[18]: df[:2]
```

```
[18]:      Decimal  Octal      Key
Code
U+0000      0    001      NUL
U+0001      1    002  Ctrl-A
```

```
[19]: df[df["Decimal"] > 2]
```

```
[19]:      Decimal  Octal      Key
Code
U+0003      3    004  Ctrl-C
U+0004      4    004  Ctrl-D
U+0005      5    005  Ctrl-E
```

The line selection syntax `df[:2]` is provided for convenience. Passing a single item or a list to the `[]` operator selects columns.

Another use case is indexing with a Boolean DataFrame, which is generated by a scalar comparison, for example:

```
[19]: df[df["Decimal"] > 2]
```

```
[19]: Code
U+0000  False
U+0001  False
U+0002  False
U+0003   True
U+0004   True
U+0005   True
Name: Decimal, dtype: bool
```

```
[20]: df[df["Decimal"] > 2] = "NA"
```

```
df
```

```
[20]:      Decimal  Octal      Key
Code
U+0000      0    001      NUL
U+0001      1    002  Ctrl-A
U+0002      2    003  Ctrl-B
U+0003     NA     NA      NA
U+0004     NA     NA      NA
U+0005     NA     NA      NA
```


Like Series, DataFrame has special operators `loc` and `iloc` for label-based and integer indexing respectively. Since DataFrame is two-dimensional, you can select a subset of the rows and columns with NumPy-like notation using either axis labels (`loc`) or integers (`iloc`).

```
[21]: df.loc["U+0002", ["Decimal", "Key"]]
```

```
[21]: Decimal      2
      Key      Ctrl-B
      Name: U+0002, dtype: object
```

```
[22]: df.iloc[[2], [1, 2]]
```

```
[22]:      Octal      Key
      Code
      U+0002  003  Ctrl-B
```

```
[23]: df.iloc[[0, 1], [1, 2]]
```

```
[23]:      Octal      Key
      Code
      U+0000  001      NUL
      U+0001  002  Ctrl-A
```

Both indexing functions work with slices in addition to individual labels or lists of labels:

```
[24]: df.loc[:"U+0003", "Key"]
```

```
[24]: Code
      U+0000      NUL
      U+0001  Ctrl-A
      U+0002  Ctrl-B
      U+0003      NA
      Name: Key, dtype: object
```

```
[25]: df.iloc[:3, :3]
```

```
[25]:      Decimal Octal      Key
      Code
      U+0000      0  001      NUL
      U+0001      1  002  Ctrl-A
      U+0002      2  003  Ctrl-B
```

So there are many ways to select and rearrange the data contained in a pandas object. In the following, I put together a brief summary of most of these possibilities for DataFrames:

Type	Note
<code>df[LABEL]</code>	selects a single column or a sequence of columns from the DataFrame
<code>df.loc[LABEL]</code>	selects a single row or a subset of rows from the DataFrame by label
<code>df.loc[:, LABEL]</code>	selects a single column or a subset of columns from the DataFrame by Label
<code>df.loc[LABEL1, LABEL2]</code>	selects both rows and columns by label
<code>df.iloc[INTEGER]</code>	selects a single row or a subset of rows from the DataFrame by integer position
<code>df.iloc[INTEGER1, INTEGER2]</code>	selects a single column or a subset of columns by integer position
<code>df.at[LABEL1, LABEL2]</code>	selects a single value by row and column label
<code>df.iat[INTEGER1, INTEGER2]</code>	selects a scalar value by row and column position (integers)
<code>reindex NEW_INDEX</code>	selects rows or columns by label
<code>get_value, set_value</code>	deprecated since version 0.21.0: use <code>.at[]</code> or <code>.iat[]</code> instead.

2.4.6 Add, change and delete data

For many data sets, you may want to perform a transformation based on the values in an array, series or column in a DataFrame. For this, we look at the first Unicode characters:

```
[1]: import numpy as np
import pandas as pd
```

```
[2]: df = pd.DataFrame(
    {
        "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
        "Decimal": [0, 1, 2, 3, 4, 5],
        "Octal": ["001", "002", "003", "004", "004", "005"],
        "Key": ["NUL", "Ctrl-A", "Ctrl-B", "Ctrl-C", "Ctrl-D", "Ctrl-E"],
    }
)

df
```

```
[2]:
```

	Code	Decimal	Octal	Key
0	U+0000	0	001	NUL
1	U+0001	1	002	Ctrl-A
2	U+0002	2	003	Ctrl-B
3	U+0003	3	004	Ctrl-C
4	U+0004	4	004	Ctrl-D
5	U+0005	5	005	Ctrl-E

Add data

Suppose you want to add a column where the characters are assigned to the C0 or C1 control code:

```
[3]: control_code = {
    "u+0000": "C0",
    "u+0001": "C0",
    "u+0002": "C0",
    "u+0003": "C0",
    "u+0004": "C0",
```

(continues on next page)

(continued from previous page)

```
"u+0005": "C0",
}
```

The `map` method for a series accepts a function or dict-like object that contains an assignment, but here we have a small problem because some of the codes in `control_code` are lower case, but not in our DataFrame. Therefore, we need to convert each value to lower case using the `str.lower` method:

```
[4]: lowercased = df["Code"].str.lower()
```

```
lowercased
```

```
[4]: 0    u+0000
     1    u+0001
     2    u+0002
     3    u+0003
     4    u+0004
     5    u+0005
     Name: Code, dtype: object
```

```
[5]: df["Control code"] = lowercased.map(control_code)
```

```
df
```

```
[5]:
```

	Code	Decimal	Octal	Key	Control code
0	U+0000	0	001	NUL	C0
1	U+0001	1	002	Ctrl-A	C0
2	U+0002	2	003	Ctrl-B	C0
3	U+0003	3	004	Ctrl-C	C0
4	U+0004	4	004	Ctrl-D	C0
5	U+0005	5	005	Ctrl-E	C0

We could also have passed a function that does all the work:

```
[6]: df["Code"].map(lambda x: control_code[x.lower()])
```

```
[6]: 0    C0
     1    C0
     2    C0
     3    C0
     4    C0
     5    C0
     Name: Code, dtype: object
```

Using `map` is a convenient way to perform element-wise transformations and other data cleaning operations.

Change data

Note:

Replacing missing values is described in *Managing missing data with pandas*.

```
[7]: pd.Series(["Manpower", "man-made"]).str.replace("Man", "Personal", regex=False)
```

```
[7]: 0    Personalpower  
     1         man-made  
     dtype: object
```

```
[8]: pd.Series(["Man-Power", "man-made"]).str.replace("[Mm]an", "Personal", regex=True)
```

```
[8]: 0    Personal-Power  
     1    Personal-made  
     dtype: object
```

Note:

The `replace` method differs from `str.replace` in that it replaces strings element by element.

Delete data

Deleting one or more entries from an axis is easy if you already have an index array or a list without these entries.

To delete duplicates, see *Deduplicating data*.

Since this may require a bit of set theory, we return the drop method as a new object without the deleted values:

```
[9]: rng = np.random.default_rng()  
     s = pd.Series(rng.normal(size=7))
```

```
s
```

```
[9]: 0    -0.800629  
     1    -1.018902  
     2    -0.183417  
     3    -0.789888  
     4    -1.898217  
     5    -0.774574  
     6    -0.370043  
     dtype: float64
```

```
[10]: new = s.drop(2)
```

```
new
```

```
[10]: 0    -0.800629  
     1    -1.018902  
     3    -0.789888  
     4    -1.898217  
     5    -0.774574  
     6    -0.370043  
     dtype: float64
```

```
[11]: new = s.drop([2, 3])
```

```
new
```

```
[11]: 0    -0.800629
      1    -1.018902
      4    -1.898217
      5    -0.774574
      6    -0.370043
      dtype: float64
```

With DataFrames, index values can be deleted on both axes. To illustrate this, we first create an example DataFrame:

```
[12]: data = {
      "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
      "Decimal": [0, 1, 2, 3, 4, 5],
      "Octal": ["001", "002", "003", "004", "004", "005"],
      "Key": ["NUL", "Ctrl-A", "Ctrl-B", "Ctrl-C", "Ctrl-D", "Ctrl-E"],
    }
```

```
df = pd.DataFrame(data)
```

```
df
```

```
[12]:
```

	Code	Decimal	Octal	Key
0	U+0000	0	001	NUL
1	U+0001	1	002	Ctrl-A
2	U+0002	2	003	Ctrl-B
3	U+0003	3	004	Ctrl-C
4	U+0004	4	004	Ctrl-D
5	U+0005	5	005	Ctrl-E

```
[13]: df.drop([0, 1])
```

```
[13]:
```

	Code	Decimal	Octal	Key
2	U+0002	2	003	Ctrl-B
3	U+0003	3	004	Ctrl-C
4	U+0004	4	004	Ctrl-D
5	U+0005	5	005	Ctrl-E

You can also remove values from the columns by passing `axis=1` or `axis='columns'`:

```
[14]: df.drop("Decimal", axis=1)
```

```
[14]:
```

	Code	Octal	Key
0	U+0000	001	NUL
1	U+0001	002	Ctrl-A
2	U+0002	003	Ctrl-B
3	U+0003	004	Ctrl-C
4	U+0004	004	Ctrl-D
5	U+0005	005	Ctrl-E

Many functions such as `drop` that change the size or shape of a row or DataFrame can manipulate an object in place without returning a new object:

```
[15]: df.drop(0, inplace=True)
```

```
df
```

```
[15]:
```

	Code	Decimal	Octal	Key
1	U+0001	1	002	Ctrl-A
2	U+0002	2	003	Ctrl-B
3	U+0003	3	004	Ctrl-C
4	U+0004	4	004	Ctrl-D
5	U+0005	5	005	Ctrl-E

Warning:

Be careful with the `inplace` function, as the data will be irretrievably deleted.

2.4.7 Manipulation of strings

pandas offers the possibility to concisely apply Python's string methods and regular expressions to whole arrays of data.

See also:

- [string](#)
- [re](#)

Vectorised string functions in pandas

Cleaning up a cluttered dataset for analysis often requires a lot of string manipulation. To make matters worse, a column containing strings sometimes has missing data:

```
[1]: import numpy as np
import pandas as pd

addresses = {
    "Veit": np.nan,
    "Veit Schiele": "veit.schiele@cusy.io",
    "cusy GmbH": "info@cusy.io",
}
addresses = pd.Series(addresses)
```

```
addresses
```

```
[1]: Veit      NaN
Veit Schiele  veit.schiele@cusy.io
cusy GmbH    info@cusy.io
dtype: object
```

```
[2]: addresses.isna()
```

```
[2]: Veit      True
Veit Schiele False
cusy GmbH    False
dtype: bool
```

You can apply string and regular expression methods to any value (by passing a lambda or other function) using `data.map`, but this fails for NA values. To deal with this, `Series` has array-oriented methods for string operations that skip and pass NA values. These are accessed via `Series.str` attribute; for example, we could use `str.contains` to check whether each email address contains `veit`:

```
[3]: addresses.str.contains("veit")
```

```
[3]: Veit          NaN
     Veit Schiele    True
     cusy GmbH      False
     dtype: object
```

Regular expressions can also be used, along with options such as `IGNORECASE`:

```
[4]: import re
```

```
pattern = r"([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})"
matches = addresses.str.findall(pattern, flags=re.IGNORECASE).str[0]

matches
```

```
[4]: Veit          NaN
     Veit Schiele  (veit.schiele, cusy, io)
     cusy GmbH    (info, cusy, io)
     dtype: object
```

There are several ways to retrieve a vectorised element. Either use `str.get` or the index of `str`:

```
[5]: matches.str.get(1)
```

```
[5]: Veit          NaN
     Veit Schiele  cusy
     cusy GmbH    cusy
     dtype: object
```

Similarly, you can also cut strings with this syntax:

```
[6]: addresses.str[:5]
```

```
[6]: Veit          NaN
     Veit Schiele  veit.
     cusy GmbH    info@
     dtype: object
```

The `pandas.Series.str.extract` method returns the captured groups of a regular expression as a `DataFrame`:

```
[7]: addresses.str.extract(pattern, flags=re.IGNORECASE)
```

```
[7]:           0      1      2
     Veit      NaN   NaN   NaN
     Veit Schiele  veit.schiele  cusy  io
     cusy GmbH    info  cusy  io
```

More vectorised pandas string methods:

Method	Description
<code>cat</code>	concatenates strings element by element with optional delimiter
<code>contains</code>	returns a boolean array if each string contains a pattern/regex
<code>count</code>	counts occurrences of the pattern
<code>extract</code>	uses a regular expression with groups to extract one or more strings from a set of strings; the result is a DataFrame with one column per group
<code>endswith</code>	equivalent to <code>x.endswith(pattern)</code> for each element
<code>startswith</code>	equivalent to <code>x.startswith(pattern)</code> for each element
<code>findall</code>	computes list of all occurrences of pattern/regex for each string
<code>get</code>	index in each element (get <i>i</i> -th element)
<code>isalnum</code>	Equivalent to built-in <code>str.alnum</code>
<code>isalpha</code>	Equivalent to built-in <code>str.isalpha</code>
<code>isdecimal</code>	Equivalent to built-in <code>str.isdecimal</code>
<code>isdigit</code>	Equivalent to built-in <code>str.isdigit</code>
<code>islower</code>	Equivalent to built-in <code>str.islower</code>
<code>isnumeric</code>	Equivalent to built-in <code>str.isnumeric</code>
<code>isupper</code>	Equivalent to built-in <code>str.isupper</code>
<code>join</code>	joins strings in each element of the series with the passed separator character
<code>len</code>	calculates the length of each string
<code>lower</code> , <code>upper</code>	converts case; equivalent to <code>x.lower()</code> or <code>x.upper()</code> for each element
<code>match</code>	uses <code>re.match</code> with the passed regular expression for each element, returning True or False if matched.
<code>extract</code>	captures group elements (if any) by index from each string
<code>pad</code>	inserts spaces on the left, right or both sides of strings
<code>centre</code>	Equivalent to <code>pad(side='both')</code>
<code>repeat</code>	Duplicates values (for example <code>s.str.repeat(3)</code> equals <code>x * 3</code> for each string)
<code>replace</code>	replaces pattern/regex with another string
<code>slice</code>	splits each string in the series
<code>split</code>	splits strings using delimiters or regular expressions
<code>strip</code>	truncates spaces on both sides, including line breaks
<code>rstrip</code>	truncates spaces on the right side
<code>lstrip</code>	truncates spaces on the left side

2.4.8 Arithmetic

An important function of pandas is the arithmetic behaviour for objects with different indices. When adding objects, if the index pairs are not equal, the corresponding index in the result will be the union of the index pairs. For users with database experience, this is comparable to an automatic `outer join` on the index labels. Let's look at an example:

```
[1]: import numpy as np
import pandas as pd

rng = np.random.default_rng()
s1 = pd.Series(rng.normal(size=5))
s2 = pd.Series(rng.normal(size=7))
```

If you add these values, you get:


```
[2]: s1 + s2
[2]: 0    2.596929
     1   -2.795545
     2   -0.119064
     3    0.849508
     4   -0.061194
     5         NaN
     6         NaN
dtype: float64
```

The internal data matching leads to missing values at the points of the labels that do not overlap. Missing values are then passed on in further arithmetic calculations.

For DataFrames, alignment is performed for both rows and columns:

```
[3]: df1 = pd.DataFrame(rng.normal(size=(5,3)))
     df2 = pd.DataFrame(rng.normal(size=(7,2)))
```

When the two DataFrames are added together, the result is a DataFrame whose index and columns are the unions of those in each of the DataFrames above:

```
[4]: df1 + df2
[4]:      0      1  2
0 -0.078026  0.643059 NaN
1 -0.383531  2.018909 NaN
2 -2.770130 -0.751184 NaN
3 -0.679346  0.926763 NaN
4 -1.093289  1.424987 NaN
5         NaN         NaN NaN
6         NaN         NaN NaN
```

Since column 2 does not appear in both DataFrame objects, its values appear as missing in the result. The same applies to the rows whose labels do not appear in both objects.

Arithmetic methods with fill values

In arithmetic operations between differently indexed objects, a special value (e.g. 0) can be useful if an axis label is found in one object but not in the other. The add method can pass the `fill_value` argument:

```
[5]: df12 = df1.add(df2, fill_value=0)
df12
[5]:      0      1  2
0 -0.078026  0.643059  0.136076
1 -0.383531  2.018909 -0.660599
2 -2.770130 -0.751184 -1.709924
3 -0.679346  0.926763 -1.403627
4 -1.093289  1.424987 -0.283248
5  0.030022 -1.465972      NaN
6 -0.508131  0.527970      NaN
```

In the following example, we set the two remaining NaN values to 0:

```
[6]: df12.iloc[[5, 6], [2]] = 0
```

```
[7]: df12
```

```
[7]:
```

	0	1	2
0	-0.078026	0.643059	0.136076
1	-0.383531	2.018909	-0.660599
2	-2.770130	-0.751184	-1.709924
3	-0.679346	0.926763	-1.403627
4	-1.093289	1.424987	-0.283248
5	0.030022	-1.465972	0.000000
6	-0.508131	0.527970	0.000000

Arithmetic methods

Method	Description
add, radd	methods for addition (+)
sub, rsub	methods for subtraction (-)
div, rdiv	methods for division (/)
floordiv, rfloordiv	methods for floor division (//)
mul, rmul	methods for multiplication (*)
pow, rpow	methods for exponentiation (**)

r (English: *reverse*) reverses the method.

Operations between DataFrame and Series

As with NumPy arrays of different dimensions, the arithmetic between DataFrame and Series is also defined.

```
[8]: s1 + df12
```

```
[8]:
```

	0	1	2	3	4
0	0.583883	-1.140178	0.991236	NaN	NaN
1	0.278378	0.235672	0.194562	NaN	NaN
2	-2.108221	-2.534422	-0.854764	NaN	NaN
3	-0.017437	-0.856475	-0.548466	NaN	NaN
4	-0.431380	-0.358250	0.571912	NaN	NaN
5	0.691931	-3.249210	0.855161	NaN	NaN
6	0.153778	-1.255268	0.855161	NaN	NaN

If we add s1 with df12, the addition is done once for each line. This is called *broadcasting*. By default, the arithmetic between the DataFrame and the series corresponds to the index of the series in the columns of the DataFrame, with the rows being broadcast down.

If an index value is found neither in the columns of the DataFrame nor in the index of the series, the objects are re-indexed to form the union:

If instead you want to transfer the columns and match the rows, you must use one of the arithmetic methods, for example:

```
[9]: df12.add(s2, axis="index")
```

```
[9]:
```

	0	1	2
0	1.856994	2.578079	2.071096
1	-1.395838	1.006602	-1.672906
2	-3.744354	-1.725408	-2.684148
3	-0.239294	1.366814	-0.963576
4	-1.067525	1.450751	-0.257484
5	0.005172	-1.490822	-0.024850
6	-0.612072	0.424029	-0.103941

The axis number you pass is the axis to be aligned to. In this case, the row index of the DataFrame (`axis='index'` or `axis=0`) is to be adjusted and transmitted.

Function application and mapping

`numpy.ufunc` (element-wise array methods) also work with pandas objects:

```
[10]: np.abs(df12)
```

```
[10]:
```

	0	1	2
0	0.078026	0.643059	0.136076
1	0.383531	2.018909	0.660599
2	2.770130	0.751184	1.709924
3	0.679346	0.926763	1.403627
4	1.093289	1.424987	0.283248
5	0.030022	1.465972	0.000000
6	0.508131	0.527970	0.000000

Another common operation is to apply a function to one-dimensional arrays on each column or row. The `pandas.DataFrame.apply` method does just that:

```
[11]: df12
```

```
[11]:
```

	0	1	2
0	-0.078026	0.643059	0.136076
1	-0.383531	2.018909	-0.660599
2	-2.770130	-0.751184	-1.709924
3	-0.679346	0.926763	-1.403627
4	-1.093289	1.424987	-0.283248
5	0.030022	-1.465972	0.000000
6	-0.508131	0.527970	0.000000

```
[12]: f = lambda x: x.max() - x.min()
```

```
df12.apply(f)
```

```
[12]:
```

0	2.800152
1	3.484882
2	1.846000

```
dtype: float64
```

Here the function `f`, which calculates the difference between the maximum and minimum of a row, is called once for each column of the frame. The result is a row with the columns of the frame as index.

If you pass `axis='columns'` to `apply`, the function will be called once per line instead:

```
[13]: df12.apply(f, axis="columns")
```

```
[13]: 0    0.721086
      1    2.679508
      2    2.018946
      3    2.330389
      4    2.518277
      5    1.495994
      6    1.036101
      dtype: float64
```

Many of the most common array statistics (such as `sum` and `mean`) are `DataFrame` methods, so the use of `apply` is not necessary.

The function passed to `apply` does not have to return a single value; it can also return a series with multiple values:

```
[14]: def f(x):
      return pd.Series([x.min(), x.max()], index=["min", "max"])
```

```
df12.apply(f)
```

```
[14]:      0      1      2
min -2.770130 -1.465972 -1.709924
max  0.030022  2.018909  0.136076
```

You can also use element-wise Python functions. Suppose you want to round each floating point value in `df12` to two decimal places, you can do this with `pandas.DataFrame.applymap`:

```
[15]: f = lambda x: round(x, 2)
```

```
df12.applymap(f)
```

```
[15]:      0      1      2
0 -0.08  0.64  0.14
1 -0.38  2.02 -0.66
2 -2.77 -0.75 -1.71
3 -0.68  0.93 -1.40
4 -1.09  1.42 -0.28
5  0.03 -1.47  0.00
6 -0.51  0.53  0.00
```

The reason for the name `applymap` is that `Series` has a `map` method for applying an element-wise function:

```
[16]: df12[2].map(f)
```

```
[16]: 0    0.14
      1   -0.66
      2   -1.71
      3   -1.40
      4   -0.28
      5    0.00
      6    0.00
      Name: 2, dtype: float64
```

2.4.9 Descriptive statistics

pandas objects are equipped with a number of common mathematical and statistical methods. Most of them fall into the category of reductions or summary statistics, methods that extract a single value (such as the sum or mean) from a series or set of values from the rows or columns of a DataFrame. Compared to similar methods found in NumPy arrays, they also handle missing data.

```
[1]: import numpy as np
import pandas as pd

rng = np.random.default_rng()
df = pd.DataFrame(
    rng.normal(size=(7, 3)), index=pd.date_range("2022-02-02", periods=7)
)
new_index = pd.date_range("2022-02-03", periods=7)
df2 = df.reindex(new_index)

df2
```

```
[1]:
```

	0	1	2
2022-02-03	0.686507	1.870769	-0.699365
2022-02-04	-1.462243	0.833043	0.423066
2022-02-05	0.227436	-1.146793	-0.495678
2022-02-06	0.404523	0.517117	-1.475375
2022-02-07	2.022298	-0.263188	-0.478148
2022-02-08	-0.056213	0.913033	-0.723379
2022-02-09	NaN	NaN	NaN

Calling the `pandas.DataFrame.sum` method returns a series containing column totals:

```
[2]: df2.sum()

[2]: 0    1.822307
1    2.723981
2   -3.448879
dtype: float64
```

Passing `axis='columns'` or `axis=1` instead sums over the columns:

```
[3]: df2.sum(axis="columns")

[3]: 2022-02-03    1.857911
2022-02-04   -0.206135
2022-02-05   -1.415035
2022-02-06   -0.553735
2022-02-07    1.280962
2022-02-08    0.133441
2022-02-09    0.000000
Freq: D, dtype: float64
```

If an entire row or column contains all NA values, the sum is 0. This can be disabled with the `skipna` option:

```
[4]: df2.sum(axis="columns", skipna=False)
```

```
[4]: 2022-02-03    1.857911
      2022-02-04   -0.206135
      2022-02-05   -1.415035
      2022-02-06   -0.553735
      2022-02-07    1.280962
      2022-02-08    0.133441
      2022-02-09         NaN
      Freq: D, dtype: float64
```

Some aggregations, such as mean, require at least one non-NaN value to obtain a valuable result:

```
[5]: df2.mean(axis="columns")

[5]: 2022-02-03    0.619304
      2022-02-04   -0.068712
      2022-02-05   -0.471678
      2022-02-06   -0.184578
      2022-02-07    0.426987
      2022-02-08    0.044480
      2022-02-09         NaN
      Freq: D, dtype: float64
```

Options for reduction methods

Method	Description
axis	the axis of values to reduce: 0 for the rows of the DataFrame and 1 for the columns
skipna	exclude missing values; by default True.
level	reduce grouped by level if the axis is hierarchically indexed (MultiIndex)

Some methods, such as `idxmin` and `idxmax`, provide indirect statistics such as the index value at which the minimum or maximum value is reached:

```
[6]: df2.idxmax()

[6]: 0    2022-02-07
      1    2022-02-03
      2    2022-02-04
      dtype: datetime64[ns]
```

Other methods are accumulations:

```
[7]: df2.cumsum()

[7]:           0           1           2
2022-02-03  0.686507  1.870769 -0.699365
2022-02-04 -0.775736  2.703812 -0.276300
2022-02-05 -0.548300  1.557019 -0.771977
2022-02-06 -0.143777  2.074136 -2.247352
2022-02-07  1.878520  1.810948 -2.725500
2022-02-08  1.822307  2.723981 -3.448879
2022-02-09         NaN         NaN         NaN
```

Another type of method is neither reductions nor accumulations. `describe` is one such example that produces several summary statistics in one go:

```
[8]: df2.describe()
```

```
[8]:
```

	0	1	2
count	6.000000	6.000000	6.000000
mean	0.303718	0.453997	-0.574813
std	1.128201	1.043312	0.609912
min	-1.462243	-1.146793	-1.475375
25%	0.014699	-0.068112	-0.717375
50%	0.315979	0.675080	-0.597521
75%	0.616011	0.893035	-0.482530
max	2.022298	1.870769	0.423066

For non-numeric data, `describe` generates alternative summary statistics:

```
[9]: data = {
      "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
      "Octal": ["001", "002", "003", "004", "004", "005"],
    }
df3 = pd.DataFrame(data)

df3.describe()
```

```
[9]:
```

	Code	Octal
count	6	6
unique	6	5
top	U+0000	004
freq	1	2

Descriptive and summary statistics:

Method	Description
count	number of non-NA values
describe	calculation of a set of summary statistics for series or each DataFrame column
min, max	calculation of minimum and maximum values
argmin, argmax	calculation of the index points (integers) at which the minimum or maximum value was reached
idxmin, idxmax	calculation of the index labels at which the minimum or maximum values were reached
quantile	calculation of the sample quantile in the range from 0 to 1
sum	sum of the values
mean	arithmetic mean of the values
median	arithmetic median (50% quantile) of the values
mad	mean absolute deviation from the mean value
prod	product of all values
var	sample variance of the values
std	sample standard deviation of the values
skew	sample skewness (third moment) of the values
kurt	sample kurtosis (fourth moment) of the values
cumsum	cumulative sum of the values
cummin, cummax	cumulated minimum and maximum of the values respectively
cumprod	cumulated product of the values
diff	calculation of the first arithmetic difference (useful for time series)
pct_change	calculation of the percentage changes

ydata-profiling

`ydata-profiling` generates profile reports from a pandas DataFrame. The pandas `df.describe()` function is handy, but a bit basic for exploratory data analysis. `ydata-profiling` extends pandas DataFrame with `df.profile_report()`, which automatically generates a standardised report for understanding the data.

Installation

Example

```
[10]: from ydata_profiling import ProfileReport

profile = ProfileReport(df2, title="pandas Profiling Report")

profile.to_widgets()

Summarize dataset:  0%|          | 0/5 [00:00<?, ?it/s]
Generate report structure:  0%|          | 0/1 [00:00<?, ?it/s]
Render widgets:  0%|          | 0/1 [00:00<?, ?it/s]
VBox(children=(Tab(children=(Tab(children=(GridBox(children=(VBox(children=(GridspecLayout(children=(HT
```

Configuration for large datasets

By default, ydata-profiling summarises the dataset to provide the most insights for data analysis. If the computation time of profiling becomes a bottleneck, pandas-profiling offers several alternatives to overcome it. For the following examples, we first read a larger data set into pandas:

```
[11]: titanic = pd.read_csv(
        "https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"
    )
```

1. minimal mode

ydata-profiling contains a minimal configuration file `config_minimal.yaml`, in which the most expensive calculations are turned off by default. This is the recommended starting point for larger data sets.

```
[12]: profile = ProfileReport(
        titanic, title="Minimal pandas Profiling Report", minimal=True
    )

profile.to_widgets()

Summarize dataset:  0%|          | 0/5 [00:00<?, ?it/s]
Generate report structure:  0%|          | 0/1 [00:00<?, ?it/s]
Render widgets:  0%|          | 0/1 [00:00<?, ?it/s]
VBox(children=(Tab(children=(Tab(children=(GridBox(children=(VBox(children=(GridspecLayout(children=(HT
```

Further details on settings and configuration can be found in [Available settings](#).

2. Sample

An alternative option for very large data sets is to use only a part of them for the profiling report:

```
[13]: sample = titanic.sample(frac=0.05)

profile = ProfileReport(sample, title="Sample pandas Profiling Report")

profile.to_widgets()

Summarize dataset:  0%|          | 0/5 [00:00<?, ?it/s]
Generate report structure:  0%|          | 0/1 [00:00<?, ?it/s]
Render widgets:  0%|          | 0/1 [00:00<?, ?it/s]
VBox(children=(Tab(children=(Tab(children=(GridBox(children=(VBox(children=(GridspecLayout(children=(HT
```

3. Deactivate expensive calculations

To reduce the computational effort in large datasets, but still get some interesting information, some calculations can be filtered only for certain columns:

```
[14]: profile = ProfileReport()
profile.config.interactions.targets = ["Sex", "Age"]
profile.df = titanic

profile.to_widgets()

Summarize dataset:  0%|          | 0/5 [00:00<?, ?it/s]
Generate report structure:  0%|          | 0/1 [00:00<?, ?it/s]
Render widgets:  0%|          | 0/1 [00:00<?, ?it/s]
VBox(children=(Tab(children=(Tab(children=(GridBox(children=(VBox(children=(GridspecLayout(children=(HT
```

The setting `interactions.targets`, can be changed via configuration files as well as via environment variables; see [Interactions](#) for details.

4 Concurrency

Currently work is being done on a scalable Spark backend for pandas-profiling, see [Spark Profiling Development](#).

2.4.10 Sorting and ranking

Sorting a record by a criterion is another important built-in function. Sorting lexicographically by row or column index is already described in the section *Reordering and sorting from levels*. In the following we look at sorting the values with `DataFrame.sort_values` and `Series.sort_values`:

```
[1]: import numpy as np
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
rng = np.random.default_rng()
s = pd.Series(rng.normal(size=7))

s.sort_index(ascending=False)
```

```
[1]: 6    -0.521271
      5    -0.228255
      4    -1.131139
      3    -0.531495
      2     0.783785
      1    -0.311396
      0     0.088381
dtype: float64
```

All missing values are sorted to the end of the row by default:

```
[2]: s = pd.Series(rng.normal(size=7))
      s[s < 0] = np.nan

      s.sort_values()
```

```
[2]: 6     0.303859
      4     0.435222
      5     0.936456
      3     1.312848
      2     1.840338
      0          NaN
      1          NaN
dtype: float64
```

With a DataFrame you can sort on both axes. With by you specify which column or row is to be sorted:

```
[3]: df = pd.DataFrame(rng.normal(size=(7, 3)))

      df.sort_values(by=2, ascending=False)
```

```
[3]:      0         1         2
3  1.489694  0.104105  0.870251
6 -0.649611 -1.035134  0.515880
5 -0.176371  1.261471  0.242477
0  0.252096 -0.315417 -1.000917
2 -1.659567 -0.139293 -1.138415
4  1.533278  0.241760 -1.252604
1  1.929005  1.032325 -2.153640
```

You can also sort rows with axis=1 and by:

```
[4]: df.sort_values(axis=1, by=[0, 1], ascending=False)
```

```
[4]:      0         1         2
0  0.252096 -0.315417 -1.000917
1  1.929005  1.032325 -2.153640
2 -1.659567 -0.139293 -1.138415
3  1.489694  0.104105  0.870251
4  1.533278  0.241760 -1.252604
```

(continues on next page)

(continued from previous page)

```
5 -0.176371  1.261471  0.242477
6 -0.649611 -1.035134  0.515880
```

Ranking

`DataFrame.rank` and `Series.rank` assign ranks from one to the number of valid data points in an array:

```
[5]: df.rank()
[5]:
```

	0	1	2
0	4.0	2.0	4.0
1	7.0	6.0	1.0
2	1.0	3.0	3.0
3	5.0	4.0	7.0
4	6.0	5.0	2.0
5	3.0	7.0	5.0
6	2.0	1.0	6.0

If ties occur in the ranking, the middle rank is usually assigned in each group.

```
[6]: df2 = pd.concat([df, df[5:]])
df2.rank()
```

```
[6]:
```

	0	1	2
0	6.0	3.0	4.0
1	9.0	7.0	1.0
2	1.0	4.0	3.0
3	7.0	5.0	9.0
4	8.0	6.0	2.0
5	4.5	8.5	5.5
6	2.5	1.5	7.5
5	4.5	8.5	5.5
6	2.5	1.5	7.5

The parameter `min`, on the other hand, assigns the smallest rank in the group:

```
[7]: df2.rank(method="min")
```

```
[7]:
```

	0	1	2
0	6.0	3.0	4.0
1	9.0	7.0	1.0
2	1.0	4.0	3.0
3	7.0	5.0	9.0
4	8.0	6.0	2.0
5	4.0	8.0	5.0
6	2.0	1.0	7.0
5	4.0	8.0	5.0
6	2.0	1.0	7.0

Other methods with rank

Method	Description
average	default: assign the average rank to each entry in the same group
min	uses the minimum rank for the whole group
max	uses the maximum rank for the whole group
first	assigns the ranks in the order in which the values appear in the data
dense	like method='min' but the ranks always increase by 1 between groups and not according to the number of same items in a group

2.4.11 Subdividing and categorising data

Continuous data is often divided into domains or otherwise grouped for analysis.

Suppose you have data on a group of people in a study that you want to divide into discrete age groups. For this, we generate a dataframe with 250 entries between 0 and 99:

```
[1]: import numpy as np
import pandas as pd

ages = np.random.randint(0, 99, 250)
df = pd.DataFrame({"Age": ages})
```

df

```
[1]:
```

	Age
0	22
1	82
2	6
3	3
4	28
..	...
245	15
246	86
247	91
248	55
249	15

```
[250 rows x 1 columns]
```

Afterwards, pandas offers us a simple way to divide the results into ten ranges with `pandas.cut`. To get only whole years, we additionally set `precision=0`:

```
[2]: cats = pd.cut(ages, 10, precision=0)
```

cats

```
[2]: [(20.0, 29.0], (78.0, 88.0], (-0.1, 10.0], (-0.1, 10.0], (20.0, 29.0], ..., (10.0, 20.0],
      ↪ (78.0, 88.0], (88.0, 98.0], (49.0, 59.0], (10.0, 20.0]]
Length: 250
Categories (10, interval[float64, right]): [(-0.1, 10.0] < (10.0, 20.0] < (20.0, 29.0] <
      ↪ (29.0, 39.0] ... (59.0, 69.0] < (69.0, 78.0] < (78.0, 88.0] < (88.0, 98.0]]
```

With `pandas.Categorical.categories` you can display the categories:

```
[3]: cats.categories
[3]: IntervalIndex([(−0.1, 10.0], (10.0, 20.0], (20.0, 29.0], (29.0, 39.0], (39.0, 49.0], (49.0, 59.0], (59.0, 69.0], (69.0, 78.0], (78.0, 88.0], (88.0, 98.0]], dtype=
↪ 'interval[float64, right]')
```

... or even just a single category:

```
[4]: cats.categories[0]
[4]: Interval(−0.1, 10.0, closed='right')
```

With `pandas.Categorical.codes` you can display an array where for each value the corresponding category is shown:

```
[5]: cats.codes
[5]: array([2, 8, 0, 0, 2, 6, 3, 9, 2, 1, 7, 0, 5, 1, 3, 6, 6, 7, 1, 9, 1, 6,
        3, 4, 3, 2, 6, 8, 5, 0, 5, 4, 0, 8, 5, 8, 3, 8, 7, 8, 6, 1, 1, 2,
        3, 4, 7, 1, 5, 9, 4, 2, 8, 2, 9, 6, 0, 9, 0, 9, 5, 0, 1, 5, 6, 5,
        3, 9, 0, 4, 2, 8, 9, 6, 5, 4, 4, 5, 6, 1, 7, 4, 1, 7, 0, 0, 1, 3,
        3, 7, 5, 1, 9, 3, 0, 1, 7, 5, 9, 5, 3, 9, 3, 6, 7, 6, 9, 9, 6, 0,
        1, 1, 3, 2, 9, 6, 0, 2, 9, 3, 8, 3, 1, 2, 7, 2, 6, 7, 9, 6, 1, 5,
        3, 3, 1, 4, 6, 9, 8, 4, 0, 4, 8, 7, 5, 5, 4, 5, 1, 5, 2, 8, 2, 6,
        0, 1, 8, 6, 7, 1, 3, 3, 3, 1, 3, 0, 6, 3, 9, 5, 9, 4, 3, 3, 0, 9,
        7, 8, 2, 4, 1, 5, 7, 8, 6, 1, 3, 1, 4, 8, 3, 0, 0, 2, 2, 8, 9, 3,
        4, 8, 4, 0, 1, 4, 9, 2, 5, 1, 1, 5, 0, 4, 7, 1, 9, 1, 7, 8, 5, 4,
        1, 7, 0, 4, 5, 0, 1, 6, 8, 0, 8, 2, 6, 0, 7, 7, 0, 2, 3, 3, 2, 0,
        4, 0, 5, 1, 8, 9, 5, 1], dtype=int8)
```

With `value_counts` we can now look at how the number is distributed among the individual areas:

```
[6]: pd.value_counts(cats)
[6]: (10.0, 20.0]    34
     (−0.1, 10.0]   29
     (29.0, 39.0]   29
     (49.0, 59.0]   26
     (88.0, 98.0]   24
     (59.0, 69.0]   23
     (39.0, 49.0]   22
     (78.0, 88.0]   22
     (20.0, 29.0]   21
     (69.0, 78.0]   20
     Name: count, dtype: int64
```

It is striking that the age ranges do not contain an equal number of years, but with 20.0, 29.0 and 69.0, 78.0 two ranges contain only 9 years. This is due to the fact that the age range only extends from 0 to 98:

```
[7]: df.min()
[7]: Age    0
     dtype: int64
```

```
[8]: df.max()
```

```
[8]: Age      98
      dtype: int64
```

With `pandas.qcut`, on the other hand, the set is divided into areas that are approximately the same size:

```
[9]: cats = pd.qcut(ages, 10, precision=0)
```

```
[10]: pd.value_counts(cats)
```

```
[10]: (24.0, 36.0]    28
      (9.0, 15.0]    26
      (53.0, 65.0]    26
      (65.0, 76.0]    26
      (-1.0, 9.0]    25
      (15.0, 24.0]    24
      (36.0, 44.0]    24
      (76.0, 88.0]    24
      (88.0, 98.0]    24
      (44.0, 53.0]    23
      Name: count, dtype: int64
```

If we want to ensure that each age group actually includes exactly ten years, we can specify this directly with `pandas.Categorical`:

```
[11]: age_groups = ["{0} - {1}".format(i, i + 9) for i in range(0, 99, 10)]
      cats = pd.Categorical(age_groups)
```

```
cats.categories
```

```
[11]: Index(['0 - 9', '10 - 19', '20 - 29', '30 - 39', '40 - 49', '50 - 59',
            '60 - 69', '70 - 79', '80 - 89', '90 - 99'],
            dtype='object')
```

For grouping we can now use `pandas.cut`. However, the number of labels must be one less than the number of edges:

```
[12]: df["Age group"] = pd.cut(df.Age, range(0, 101, 10), right=False, labels=cats)
```

```
df
```

```
[12]:
```

	Age	Age group
0	22	20 - 29
1	82	80 - 89
2	6	0 - 9
3	3	0 - 9
4	28	20 - 29
..
245	15	10 - 19
246	86	80 - 89
247	91	90 - 99
248	55	50 - 59
249	15	10 - 19

```
[250 rows x 2 columns]
```

2.4.12 Combining and merging data sets

Data contained in pandas objects can be combined in several ways:

- `pandas.merge` joins rows in DataFrames based on one or more keys. This function is familiar from SQL or other relational databases, as it implements database join operations.
- `pandas.concat` concatenates or *stacks* objects along an axis.
- The instance methods `pandas.DataFrame.combine_first` or `pandas.Series.combine_first` allow overlapping data to be joined.
- With `pandas.merge_asof` you can perform time series based window joins between DataFrame objects.

Database-like DataFrame joins

Merge or join operations combine data sets by linking rows with one or more keys. These operations are especially important in relational, SQL-based databases. The merge function in pandas is the main entry point for applying these algorithms to your data.

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

```
[2]: encoding = pd.DataFrame(
    {
        "Unicode": [
            "U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005",
        ],
        "Decimal": [0, 1, 2, 3, 4, 5],
        "Octal": ["000", "001", "002", "003", "004", "005"],
        "Key": ["NUL", "Ctrl-A", "Ctrl-B", "Ctrl-C", "Ctrl-D", "Ctrl-E"],
    }
)

update = pd.DataFrame(
    {
        "Unicode": [
            "U+0003", "U+0004", "U+0005", "U+0006", "U+0007", "U+0008", "U+0009",
        ],
        "Decimal": [3, 4, 5, 6, 7, 8, 9],
        "Octal": ["003", "004", "005", "006", "007", "008", "009"],
        "Key": [
            "Ctrl-C", "Ctrl-D", "Ctrl-E", "Ctrl-F", "Ctrl-G", "Ctrl-H", "Ctrl-I",
        ],
    }
)

encoding, update
```

```
[2]: (  Unicode  Decimal  Octal    Key
0  U+0000         0   000    NUL
1  U+0001         1   001  Ctrl-A
2  U+0002         2   002  Ctrl-B
3  U+0003         3   003  Ctrl-C
4  U+0004         4   004  Ctrl-D
5  U+0005         5   005  Ctrl-E,
```

(continues on next page)

(continued from previous page)

	Unicode	Decimal	Octal	Key
0	U+0003	3	003	Ctrl-C
1	U+0004	4	004	Ctrl-D
2	U+0005	5	005	Ctrl-E
3	U+0006	6	006	Ctrl-F
4	U+0007	7	007	Ctrl-G
5	U+0008	8	008	Ctrl-H
6	U+0009	9	009	Ctrl-I

When we call merge with these objects, we get:

```
[3]: pd.merge(encoding, update)
```

```
[3]:   Unicode  Decimal  Octal   Key
0  U+0003         3    003  Ctrl-C
1  U+0004         4    004  Ctrl-D
2  U+0005         5    005  Ctrl-E
```

By default, merge performs a so-called *inner join*; the keys in the result are the intersection or common set in both tables.

Note:

I did not specify which column to merge over. If this information is not specified, merge will use the overlapping column names as keys. However, it is good practice to specify this explicitly:

```
[4]: pd.merge(encoding, update, on="Unicode")
```

```
[4]:   Unicode  Decimal_x  Octal_x  Key_x  Decimal_y  Octal_y  Key_y
0  U+0003         3     003  Ctrl-C         3     003  Ctrl-C
1  U+0004         4     004  Ctrl-D         4     004  Ctrl-D
2  U+0005         5     005  Ctrl-E         5     005  Ctrl-E
```

If the column names are different in each object, you can specify them separately. In the following example update2 gets the key U+ and not Unicode:

```
[5]: update2 = pd.DataFrame(
    {
        "U+": [
            "U+0003", "U+0004", "U+0005", "U+0006", "U+0007", "U+0008", "U+0009",
        ],
        "Decimal": [3, 4, 5, 6, 7, 8, 9],
        "Octal": ["003", "004", "005", "006", "007", "008", "009"],
        "Key": [
            "Ctrl-C", "Ctrl-D", "Ctrl-E", "Ctrl-F", "Ctrl-G", "Ctrl-H", "Ctrl-I",
        ],
    }
)

pd.merge(encoding, update2, left_on="Unicode", right_on="U+")
```

```
[5]:   Unicode  Decimal_x  Octal_x  Key_x      U+  Decimal_y  Octal_y  Key_y
0  U+0003         3     003  Ctrl-C  U+0003         3     003  Ctrl-C
1  U+0004         4     004  Ctrl-D  U+0004         4     004  Ctrl-D
2  U+0005         5     005  Ctrl-E  U+0005         5     005  Ctrl-E
```

However, you can use `merge` not only to perform an inner join, with which the keys in the result are the intersection or common set in both tables. Other possible options are:

Option	Behaviour
<code>how='inner'</code>	uses only the key combinations observed in both tables
<code>how='left'</code>	uses all key combinations found in the left table
<code>how='right'</code>	uses all key combinations found in the right table
<code>how='outer'</code>	uses all key combinations observed in both tables together

```
[5]: pd.merge(encoding, update, on="Unicode", how="left")
```

```
[5]:   Unicode  Decimal_x  Octal_x  Key_x  Decimal_y  Octal_y  Key_y
0  U+0000         0     000    NUL         NaN     NaN     NaN
1  U+0001         1     001  Ctrl-A         NaN     NaN     NaN
2  U+0002         2     002  Ctrl-B         NaN     NaN     NaN
3  U+0003         3     003  Ctrl-C         3.0     003  Ctrl-C
4  U+0004         4     004  Ctrl-D         4.0     004  Ctrl-D
5  U+0005         5     005  Ctrl-E         5.0     005  Ctrl-E
```

```
[7]: pd.merge(encoding, update, on="Unicode", how="outer")
```

```
[7]:   Unicode  Decimal_x  Octal_x  Key_x  Decimal_y  Octal_y  Key_y
0  U+0000         0.0     000    NUL         NaN     NaN     NaN
1  U+0001         1.0     001  Ctrl-A         NaN     NaN     NaN
2  U+0002         2.0     002  Ctrl-B         NaN     NaN     NaN
3  U+0003         3.0     003  Ctrl-C         3.0     003  Ctrl-C
4  U+0004         4.0     004  Ctrl-D         4.0     004  Ctrl-D
5  U+0005         5.0     005  Ctrl-E         5.0     005  Ctrl-E
6  U+0006         NaN     NaN     NaN         6.0     006  Ctrl-F
7  U+0007         NaN     NaN     NaN         7.0     007  Ctrl-G
8  U+0008         NaN     NaN     NaN         8.0     008  Ctrl-H
9  U+0009         NaN     NaN     NaN         9.0     009  Ctrl-I
```

The join method only affects the unique key values that appear in the result.

To join multiple keys, you can pass a list of column names:

```
[6]: pd.merge(encoding, update, on=["Unicode", "Decimal", "Octal", "Key"], how="outer")
```

```
[6]:   Unicode  Decimal  Octal  Key
0  U+0000         0    000   NUL
1  U+0001         1    001  Ctrl-A
2  U+0002         2    002  Ctrl-B
3  U+0003         3    003  Ctrl-C
4  U+0004         4    004  Ctrl-D
5  U+0005         5    005  Ctrl-E
6  U+0006         6    006  Ctrl-F
7  U+0007         7    007  Ctrl-G
8  U+0008         8    008  Ctrl-H
9  U+0009         9    009  Ctrl-I
```

2.4.13 Group operations

By `groupby` is meant a process that involves one or more of the following steps:

- **Split** divides the data into groups according to certain criteria
- **Apply** applies a function independently to each group
- **Combine** combines the results in a data structure

In the first phase of the process, the data contained in a pandas object, be it a `Series`, a `DataFrame` or something else, is split into groups based on one or more keys. The division is done on a particular axis of an object. For example, a `DataFrame` can be grouped by its rows (`axis=0`) or its columns (`axis=1`). Then, a function is applied to each group to create a new value. Finally, the results of all these function applications are combined in a result object. The shape of the result object usually depends on what is done with the data.

Each grouping key can take many forms, and the keys do not all have to be of the same type:

- a list or array of values that have the same length as the axis being grouped
- a value that specifies a column name in a `DataFrame`
- a dict or series that is a correspondence between the values on the axis being grouped and the group names
- a function that is called on the axis index or the individual labels in the index

Note:

The latter three methods are shortcuts to create an array of values that will be used to divide the object.

Don't worry if this all seems abstract. Throughout this chapter I will give many examples of all these methods. For starters, here is a small table dataset as a `DataFrame`:

```
[1]: import numpy as np
import pandas as pd
```

```
[2]: df = pd.DataFrame(
    {
        "Title": [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            None,
            "Python Basics",
            "Python Basics",
        ],
        "Language": ["de", "en", "de", None, "de", "en"],
        "2021-12": [19651, 4722, 2573, None, 525, 157],
        "2022-01": [30134, 3497, 4873, None, 427, 85],
        "2022-02": [33295, 4009, 3930, None, 276, 226],
    }
)

df
```

```
[2]:
```

	Title	Language	2021-12	2022-01	2022-02
0	Jupyter Tutorial	de	19651.0	30134.0	33295.0

(continues on next page)

(continued from previous page)

1	Jupyter Tutorial	en	4722.0	3497.0	4009.0
2	PyViz Tutorial	de	2573.0	4873.0	3930.0
3	None	None	NaN	NaN	NaN
4	Python Basics	de	525.0	427.0	276.0
5	Python Basics	en	157.0	85.0	226.0

Suppose you want to calculate the sum of column 02/2022 using the labels of Title. There are several ways to do this. One is to access 02/2022 and call `groupby` with the column (a Series) in Title:

```
[3]: grouped = df["2022-02"].groupby(df["Title"])
```

```
grouped
```

```
[3]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x11f32c8d0>
```

This `grouped` variable is now a special `SeriesGroupBy` object. It has not yet calculated anything except some intermediate data about the group key `df['Title']`. The idea is that this object has all the information needed to apply an operation to each of the groups. For example, to calculate the group averages, we can call the `sum` method of the `GroupBy` object:

```
[4]: grouped.sum()
```

```
[4]: Title
Jupyter Tutorial    37304.0
PyViz Tutorial     3930.0
Python Basics      502.0
Name: 2022-02, dtype: float64
```

Later I will explain more about what happens when you call `.sum()`. The important thing to note here is that the data (a row) has been aggregated by splitting the data across the group key, creating a new row that is now indexed by the unique values in the Title column. The resulting index is Title because `groupby(df['Title'])` did this.

If we had passed multiple arrays as a list instead, we would get something different:

```
[5]: sums = df["2021-12"].groupby([df["Language"], df["Title"]]).sum()
```

```
sums
```

```
[5]: Language Title
de      Jupyter Tutorial    19651.0
        PyViz Tutorial      2573.0
        Python Basics       525.0
en      Jupyter Tutorial    4722.0
        Python Basics       157.0
Name: 2021-12, dtype: float64
```

Here we have grouped the data based on two keys, and the resulting series now has a hierarchical index consisting of the observed unique key pairs:

```
[6]: sums.unstack()
```

```
[6]: Title      Jupyter Tutorial  PyViz Tutorial  Python Basics
Language
de              19651.0           2573.0           525.0
en              4722.0             NaN           157.0
```

Often the grouping information is in the same DataFrame as the data you want to edit. In this case, you can pass column names (whether they are strings, numbers or other Python objects) as group keys:

```
[7]: df.groupby("Title").sum()
```

```
[7]:
```

	Language	2021-12	2022-01	2022-02
Title				
Jupyter Tutorial	deen	24373.0	33631.0	37304.0
PyViz Tutorial	de	2573.0	4873.0	3930.0
Python Basics	deen	682.0	512.0	502.0

Here it is noticeable that the result does not contain a Language column. Since `df['Language']` is not numeric data, it interferes with the table layout and is therefore automatically excluded from the result. By default, all numeric columns are aggregated.

```
[8]: df.groupby(["Title", "Language"]).sum()
```

```
[8]:
```

		2021-12	2022-01	2022-02
Title	Language			
Jupyter Tutorial	de	19651.0	30134.0	33295.0
	en	4722.0	3497.0	4009.0
PyViz Tutorial	de	2573.0	4873.0	3930.0
Python Basics	de	525.0	427.0	276.0
	en	157.0	85.0	226.0

Regardless of the goal of using `groupby`, a generally useful `groupby` method is `size`, which returns a series with the group sizes:

```
[9]: df.groupby(["Language"]).size()
```

```
[9]: Language
de      3
en      2
dtype: int64
```

Note:

All missing values in a group key are excluded from the result by default. This behaviour can be disabled by passing `dropna=False` to `groupby`:

```
[10]: df.groupby("Language", dropna=False).size()
```

```
[10]: Language
de      3
en      2
NaN     1
dtype: int64
```

```
[11]: df.groupby(["Title", "Language"], dropna=False).size()
```

```
[11]:
```

Title	Language	
Jupyter Tutorial	de	1
	en	1
PyViz Tutorial	de	1
Python Basics	de	1
	en	1

(continues on next page)

(continued from previous page)

```
NaN      NaN      1
dtype: int64
```

Iteration over groups

The object returned by `groupby` supports iteration and produces a sequence of 2-tuples containing the group name along with the data packet. Consider the following:

```
[12]: for name, group in df.groupby("Title"):
      print(name)
      print(group)
```

```
Jupyter Tutorial
      Title Language 2021-12 2022-01 2022-02
0  Jupyter Tutorial    de 19651.0 30134.0 33295.0
1  Jupyter Tutorial    en  4722.0  3497.0  4009.0
PyViz Tutorial
      Title Language 2021-12 2022-01 2022-02
2  PyViz Tutorial     de  2573.0  4873.0  3930.0
Python Basics
      Title Language 2021-12 2022-01 2022-02
4  Python Basics     de   525.0   427.0   276.0
5  Python Basics     en   157.0    85.0   226.0
```

With multiple keys, the first element of the tuple is a tuple of key values:

```
[13]: for (i1, i2), group in df.groupby(["Title", "Language"]):
      print((i1, i2))
      print(group)
```

```
('Jupyter Tutorial', 'de')
      Title Language 2021-12 2022-01 2022-02
0  Jupyter Tutorial    de 19651.0 30134.0 33295.0
('Jupyter Tutorial', 'en')
      Title Language 2021-12 2022-01 2022-02
1  Jupyter Tutorial    en  4722.0  3497.0  4009.0
('PyViz Tutorial', 'de')
      Title Language 2021-12 2022-01 2022-02
2  PyViz Tutorial     de  2573.0  4873.0  3930.0
('Python Basics', 'de')
      Title Language 2021-12 2022-01 2022-02
4  Python Basics     de   525.0   427.0   276.0
('Python Basics', 'en')
      Title Language 2021-12 2022-01 2022-02
5  Python Basics     en   157.0    85.0   226.0
```

Next, we want to output a dict of the data as a one-liner:

```
[14]: books = dict(list(df.groupby("Title")))

books
```

```
[14]: {'Jupyter Tutorial':
      0 Jupyter Tutorial      de 19651.0 30134.0 33295.0
      1 Jupyter Tutorial      en  4722.0  3497.0  4009.0,
      'PyViz Tutorial':
      2 PyViz Tutorial        de  2573.0  4873.0  3930.0,
      'Python Basics':
      4 Python Basics        de   525.0   427.0   276.0
      5 Python Basics        en   157.0    85.0   226.0}
```

By default, `groupby` groups on `axis=0`, but you can also group on any of the other axes. For example, we could group the columns of our example `df` here by `dtype` as follows:

```
[15]: df.dtypes
```

```
[15]: Title      object
      Language    object
      2021-12     float64
      2022-01     float64
      2022-02     float64
      dtype: object
```

```
[16]: grouped = df.groupby(df.dtypes, axis=1)
```

```
[17]: for dtype, group in grouped:
      print(dtype)
      print(group)
```

```
float64
      2021-12  2022-01  2022-02
0  19651.0    30134.0  33295.0
1   4722.0     3497.0   4009.0
2   2573.0     4873.0   3930.0
3         NaN         NaN         NaN
4    525.0      427.0    276.0
5    157.0       85.0    226.0
object
      Title Language
0  Jupyter Tutorial      de
1  Jupyter Tutorial      en
2   PyViz Tutorial      de
3             None      None
4   Python Basics      de
5   Python Basics      en
```

Selecting a column or subset of columns

Indexing a `GroupBy` object created from a `DataFrame` with a column name or an array of column names has the effect of subdividing columns for aggregation. This means that:

```
[18]: df.groupby("Title")["2021-12"]  
df.groupby("Title")[["2022-01"]]
```

```
[18]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x11f2f08d0>
```

are simplified spellings for:

```
[19]: df["2021-12"].groupby(df["Title"])  
df[["2022-01"]].groupby(df["Title"])
```

```
[19]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x11f362a50>
```

Especially for large datasets, it may be desirable to aggregate only some columns. For example, to calculate the sum for only column `01/2022` in the previous dataset and get the result as a `DataFrame`, we could write:

```
[20]: df.groupby(["Title", "Language"])["2022-01"].sum()
```

```
[20]:
```

		2022-01
Title	Language	
Jupyter Tutorial	de	30134.0
	en	3497.0
PyViz Tutorial	de	4873.0
Python Basics	de	427.0
	en	85.0

The object returned by this indexing operation is a grouped `DataFrame` if a list or array is passed, or a grouped series if only a single column name is passed as a scalar:

```
[21]: series_grouped = df.groupby(["Title", "Language"])["2022-01"]
```

```
series_grouped
```

```
[21]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x11f36d150>
```

```
[22]: series_grouped.sum()
```

```
[22]:
```

Title	Language	
Jupyter Tutorial	de	30134.0
	en	3497.0
PyViz Tutorial	de	4873.0
Python Basics	de	427.0
	en	85.0

```
Name: 2022-01, dtype: float64
```


Grouping with dicts and series

Grouping information can also be in a form other than an array:

```
[23]: df.iloc[2:3, [2, 3]] = np.nan
```

Suppose I have a group correspondence for the columns and want to group the columns together by group:

```
[24]: mapping = {"2021-12": "Dec 2021",
                "2022-01": "Jan 2022",
                "2022-02": "Feb 2022"}
```

Now an array could be constructed from this dict to pass to `groupby`, but instead we can just pass the dict:

```
[25]: by_column = df.groupby(mapping, axis=1)
```

```
by_column.sum()
```

```
[25]:
```

	Dec 2021	Feb 2022	Jan 2022
0	19651.0	33295.0	30134.0
1	4722.0	4009.0	3497.0
2	0.0	3930.0	0.0
3	0.0	0.0	0.0
4	525.0	276.0	427.0
5	157.0	226.0	85.0

The same functionality applies to Series:

```
[26]: map_series = pd.Series(mapping)
```

```
map_series
```

```
[26]: 2021-12    Dec 2021
      2022-01    Jan 2022
      2022-02    Feb 2022
dtype: object
```

```
[27]: df.groupby(map_series, axis=1).sum()
```

```
[27]:
```

	Dec 2021	Feb 2022	Jan 2022
0	19651.0	33295.0	30134.0
1	4722.0	4009.0	3497.0
2	0.0	3930.0	0.0
3	0.0	0.0	0.0
4	525.0	276.0	427.0
5	157.0	226.0	85.0

Grouping with Functions

Using Python functions is a more general method of defining a group assignment compared to a `Dict` or `Series`. Each function passed as a group key is called once per index value, with the return values used as group names. Specifically, consider the example `DataFrame` from the previous section, which contains the titles as index values. Suppose If you want to group by the length of the names, you can calculate an array with the lengths of the strings, but it is easier to pass the `len` function:

```
[28]: df = pd.DataFrame(  
    [  
        [19651, 30134, 33295],  
        [4722, 3497, 4009],  
        [2573, 4873, 3930],  
        [525, 427, 276],  
        [157, 85, 226],  
    ],  
    index=[  
        "Jupyter Tutorial",  
        "Jupyter Tutorial",  
        "PyViz Tutorial",  
        "Python Basics",  
        "Python Basics",  
    ],  
    columns=["2021-12", "2022-01", "2022-02"],  
)
```

```
[29]: df.groupby(len).count()
```

```
[29]:
```

	2021-12	2022-01	2022-02
13	2	2	2
14	1	1	1
16	2	2	2

Mixing functions with arrays, dicts or series is no problem, as everything is converted internally into arrays:

```
[30]: languages = ["de", "en", "de", "de", "en"]
```

```
[31]: df.groupby([len, languages]).count()
```

```
[31]:
```

		2021-12	2022-01	2022-02
13	de	1	1	1
	en	1	1	1
14	de	1	1	1
16	de	1	1	1
	en	1	1	1

Grouping by index levels

A final practical feature for hierarchically indexed datasets is the ability to aggregate by one of the index levels of an axis. Let's look at an example:

```
[32]: version_hits = [
    [19651, 0, 30134, 0, 33295, 0],
    [4722, 1825, 3497, 2576, 4009, 3707],
    [2573, 0, 4873, 0, 3930, 0],
    [None, None, None, None, None, None],
    [525, 0, 427, 0, 276, 0],
    [157, 0, 85, 0, 226, 0],
]

df = pd.DataFrame(
    version_hits,
    index=[
        [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            None,
            "Python Basics",
            "Python Basics",
        ],
        ["de", "en", "de", None, "de", "en"],
    ],
    columns=[
        ["2021-12", "2021-12", "2022-01", "2022-01", "2022-02", "2022-02"],
        ["latest", "stable", "latest", "stable", "latest", "stable"],
    ],
)

df.columns.names = ["Month", "Version"]

df
```

```
[32]: Month          2021-12      2022-01      2022-02
Version latest stable latest stable latest stable
Jupyter Tutorial de  19651.0    0.0  30134.0    0.0  33295.0    0.0
                  en   4722.0  1825.0   3497.0  2576.0   4009.0  3707.0
PyViz Tutorial   de   2573.0    0.0   4873.0    0.0   3930.0    0.0
NaN              NaN      NaN      NaN      NaN      NaN      NaN
Python Basics   de    525.0    0.0    427.0    0.0    276.0    0.0
                  en    157.0    0.0     85.0    0.0    226.0    0.0
```

```
[33]: df.groupby(level="Month", axis=1).sum()

[33]: Month          2021-12  2022-01  2022-02
Jupyter Tutorial de  19651.0  30134.0  33295.0
                  en   6547.0   6073.0   7716.0
PyViz Tutorial   de   2573.0   4873.0   3930.0
NaN              NaN     0.0     0.0     0.0
Python Basics   de    525.0   427.0   276.0
```

(continues on next page)

(continued from previous page)

en	157.0	85.0	226.0
----	-------	------	-------

2.4.14 Aggregation

Aggregations refer to any data transformation that produces scalar values from arrays. In the previous examples, several of them were used, including count and sum. You may now be wondering what happens when you apply sum() to a GroupBy object. Optimised implementations exist for many common aggregations, such as the one in the following table. However, they are not limited to this set of methods.

Function name Description

Function name	Description
any, all	Returns True if one (or more) or all of the non-NA values are truthy
count	Number of non-NA values
cummin, cummax	Cumulative minimum and maximum of the non-NA values
cumsum	Cumulative sum of the non-NA values
cumprod	Cumulative product of non-NA values
first, last	First and last non-NA values
mean	Mean of the non-NA values
median	Arithmetic median of the non-NA values
min, max	Minimum and maximum of the non-NA values
nth	Retrieval of the nth largest value
ohlc	calculates the four <i>open-high-low-close</i> statistics for time series-like data
prod	Product of the non-NA values
quantile	calculates the sample quantile
rank	Ordinal ranks of non-NA values, as when calling Series.rank
sum	Sum of non-NA values
std, var	Standard deviation and variance of the sample

You can use your own aggregations and also call any method that is also defined for the grouped object. For example, the Series method nsmallest selects the smallest requested number of values from the data.

Although nsmallest is not explicitly implemented for GroupBy, we can still use it with a non-optimised implementation. Internally, GroupBy decomposes the Series, calls df.nsmallest(n) for each part and then merges these results in the result object:

```
[1]: import numpy as np
import pandas as pd

[2]: df = pd.DataFrame(
    {
        "Title": [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            None,
            "Python Basics",
            "Python Basics",
        ],
        "2021-12": [30134, 6073, 4873, None, 427, 95],
    })
```

(continues on next page)

(continued from previous page)

```

    "2022-01": [33295, 7716, 3930, None, 276, 226],
    "2022-02": [19651, 6547, 2573, None, 525, 157],
}
)

df

```

```

[2]:
      Title  2021-12  2022-01  2022-02
0  Jupyter Tutorial  30134.0  33295.0  19651.0
1  Jupyter Tutorial   6073.0   7716.0   6547.0
2   PyViz Tutorial   4873.0   3930.0   2573.0
3             None         NaN         NaN
4   Python Basics    427.0    276.0    525.0
5   Python Basics     95.0    226.0    157.0

```

```
[3]: grouped = df.groupby("Title")
```

```
[4]: grouped["2022-01"].nsmallest(1)
```

```

[4]: Title
Jupyter Tutorial    1    7716.0
PyViz Tutorial     2    3930.0
Python Basics     5     226.0
Name: 2022-01, dtype: float64

```

To use a custom aggregation function, pass any function that aggregates an array to the `aggregate` or `agg` method:

```

[5]: def range(arr):
      return arr.max() - arr.min()

```

```
grouped.agg(range)
```

```

[5]:
      Title  2021-12  2022-01  2022-02
Jupyter Tutorial  24061.0  25579.0  13104.0
PyViz Tutorial     0.0     0.0     0.0
Python Basics    332.0    50.0    368.0

```

You will find that some methods like `describe` also work, even though they are not strictly speaking aggregations:

```
[6]: grouped.describe()
```

```

[6]:
      Title  2021-12  count  mean  std  min  25%  50% \
Jupyter Tutorial    2.0  18103.5  17013.696262  6073.0  12088.25  18103.5
PyViz Tutorial     1.0   4873.0         NaN  4873.0   4873.00  4873.0
Python Basics     2.0   261.0   234.759451    95.0   178.00   261.0

      Title  2022-01  count  mean  ...  75%  max
Jupyter Tutorial  24118.75  30134.0    2.0  20505.5  ...  26900.25  33295.0
PyViz Tutorial   4873.00  4873.0    1.0  3930.0  ...  3930.00  3930.0

```

(continues on next page)

(continued from previous page)

```

Python Basics      344.00    427.0      2.0    251.0    ...    263.50    276.0

                2022-02
                count      mean      std      min      25%      50%
Title
Jupyter Tutorial    2.0   13099.0  9265.927261  6547.0  9823.0  13099.0
PyViz Tutorial      1.0   2573.0      NaN  2573.0  2573.0  2573.0
Python Basics      2.0   341.0   260.215295  157.0   249.0   341.0

                75%      max
Title
Jupyter Tutorial   16375.0  19651.0
PyViz Tutorial     2573.0   2573.0
Python Basics     433.0   525.0

[3 rows x 24 columns]

```

Note:

Custom aggregation functions are generally much slower than the optimised functions in the table above. This is because there is some extra work involved in creating the intermediate data sets for the group (function calls, reordering of data).

Additional functions column by column

As we have already seen, aggregating a `Series` or all columns of a `DataFrame` is a matter of using `aggregate` (or `agg`) with the desired function or calling a method such as `mean` or `std`. However, it is more common to aggregate simultaneously with another function depending on the column or with multiple functions.

```
[7]: grouped.agg("mean")
```

```

[7]:           2021-12  2022-01  2022-02
Title
Jupyter Tutorial  18103.5  20505.5  13099.0
PyViz Tutorial    4873.0   3930.0   2573.0
Python Basics    261.0    251.0    341.0

```

If you pass a list of functions or function names instead, you will get back a `DataFrame` with column names from the functions:

```
[8]: grouped.agg(["mean", "std", "range"])
```

```

[8]:           2021-12      std      range      2022-01      std
                mean
Title
Jupyter Tutorial  18103.5  17013.696262  24061.0  20505.5  18087.084356
PyViz Tutorial    4873.0      NaN      0.0   3930.0      NaN
Python Basics    261.0   234.759451   332.0    251.0   35.355339

                2022-02
                range      mean      std      range
Title
Jupyter Tutorial  25579.0  13099.0  9265.927261  13104.0

```

(continues on next page)

(continued from previous page)

PyViz Tutorial	0.0	2573.0	NaN	0.0
Python Basics	50.0	341.0	260.215295	368.0

Here we have passed `agg` a list of aggregation functions to be evaluated independently for the data groups.

You don't need to accept the names that `GroupBy` gives to the columns; in particular, lambda functions have the name `<lambda>`, which makes them difficult to identify. When you pass a list of tuples, the first element of each tuple is used as the column name in the `DataFrame`:

```
[9]: grouped.agg(
      [("Mean", "mean"), ("Standard deviation", "std"), ("Range", "range")]
    )
```

```
[9]:
```

	2021-12			2022-01 \	
Title	Mean	Standard deviation	Range	Mean	
Jupyter Tutorial	18103.5	17013.696262	24061.0	20505.5	
PyViz Tutorial	4873.0	NaN	0.0	3930.0	
Python Basics	261.0	234.759451	332.0	251.0	

	2022-02 \		
Title	Standard deviation	Range	Mean
Jupyter Tutorial	18087.084356	25579.0	13099.0
PyViz Tutorial	NaN	0.0	2573.0
Python Basics	35.355339	50.0	341.0

	Range
Title	
Jupyter Tutorial	13104.0
PyViz Tutorial	0.0
Python Basics	368.0

With a `DataFrame`, you have the option of specifying a list of functions to be applied to all columns or to different functions per column. Let's say we want to calculate the same three statistics for the columns:

```
[10]: stats = ["count", "mean", "max"]
```

```
evaluations = grouped.agg(stats)
```

```
evaluations
```

```
[10]:
```

	2021-12			2022-01			2022-02 \	
Title	count	mean	max	count	mean	max	count	
Jupyter Tutorial	2	18103.5	30134.0	2	20505.5	33295.0	2	
PyViz Tutorial	1	4873.0	4873.0	1	3930.0	3930.0	1	
Python Basics	2	261.0	427.0	2	251.0	276.0	2	

	mean	max
Title		
Jupyter Tutorial	13099.0	19651.0

(continues on next page)

(continued from previous page)

PyViz Tutorial	2573.0	2573.0
Python Basics	341.0	525.0

As you can see, the resulting DataFrame has hierarchical columns, just as you would get if you aggregated each column separately and used `pandas.concat` to join the results together, using the column names as key arguments:

```
[11]: evaluations["2021-12"]
```

```
[11]:
```

	count	mean	max
Title			
Jupyter Tutorial	2	18103.5	30134.0
PyViz Tutorial	1	4873.0	4873.0
Python Basics	2	261.0	427.0

As before, a list of tuples with user-defined names can be passed:

```
[12]: tuples = [("Mean", "mean"), ("Variance", np.var)]
```

```
grouped[["2021-12", "2022-01"]].agg(tuples)
```

```
[12]:
```

	2021-12		2022-01	
	Mean	Variance	Mean	Variance
Title				
Jupyter Tutorial	18103.5	289465860.5	20505.5	327142620.5
PyViz Tutorial	4873.0	NaN	3930.0	NaN
Python Basics	261.0	55112.0	251.0	1250.0

If we now assume that potentially different functions are to be applied to one or more of the columns, we pass a dict to `agg` that contains an assignment of column names to one of the function specifications:

```
[13]: grouped.agg({"2021-12": "mean", "2022-01": np.var})
```

```
[13]:
```

	2021-12	2022-01
Title		
Jupyter Tutorial	18103.5	327142620.5
PyViz Tutorial	4873.0	NaN
Python Basics	261.0	1250.0

```
[14]: grouped.agg({"2021-12": ["min", "max", "mean", "std"], "2022-01": "sum"})
```

```
[14]:
```

	2021-12				2022-01
	min	max	mean	std	sum
Title					
Jupyter Tutorial	6073.0	30134.0	18103.5	17013.696262	41011.0
PyViz Tutorial	4873.0	4873.0	4873.0	NaN	3930.0
Python Basics	95.0	427.0	261.0	234.759451	502.0

Return aggregated data without row indices

In all the examples so far, the aggregated data is returned with an index. Since this is not always desired, you can disable this behaviour in most cases by passing `as_index=False` to `groupby`:

```
[15]: grouped.agg([range], as_index=False).mean()
```

```
[15]: 2021-12  8131.000000
      2022-01  8543.000000
      2022-02  4490.666667
      dtype: float64
```

By using the method `as_index=False`, some unnecessary calculations are avoided. Of course, it is always possible to get the result back with index by calling `reset_index` for the result.

2.4.15 Apply

The most general `GroupBy` method is `apply`. It splits the object to be processed, calls the passed function on each part and then tries to chain the parts together.

Suppose we want to select the five largest `hit` values by group. To do this, we first write a function that selects the rows with the largest values in a particular column:

```
[1]: import numpy as np
      import pandas as pd
```

```
[2]: df = pd.DataFrame(
      {
          "2021-12": [30134, 6073, 4873, None, 427, 95],
          "2022-01": [33295, 7716, 3930, None, 276, 226],
          "2022-02": [19651, 6547, 2573, None, 525, 157],
      },
      index=[
          "Jupyter Tutorial",
          "Jupyter Tutorial",
          "PyViz Tutorial",
          "PyViz Tutorial",
          "Python Basics",
          "Python Basics",
      ],
      ["de", "en", "de", "en", "de", "en"],
  )
  df.index.names = ["Title", "Language"]

  df
```

```
[2]:
```

		2021-12	2022-01	2022-02
Title	Language			
Jupyter Tutorial	de	30134.0	33295.0	19651.0
	en	6073.0	7716.0	6547.0
PyViz Tutorial	de	4873.0	3930.0	2573.0
	en	NaN	NaN	NaN

(continues on next page)

(continued from previous page)

Python Basics	de	427.0	276.0	525.0
	en	95.0	226.0	157.0

```
[3]: def top(df, n=5, column="2021-12"):
      return df.sort_values(by=column, ascending=False)[:n]
```

```
top(df, n=3)
```

```
[3]:
```

		2021-12	2022-01	2022-02
Title	Language			
Jupyter Tutorial	de	30134.0	33295.0	19651.0
	en	6073.0	7716.0	6547.0
PyViz Tutorial	de	4873.0	3930.0	2573.0

If we now group by titles, for example, and call `apply` with this function, we get the following:

```
[4]: grouped_titles = df.groupby("Title", as_index=False)
```

```
grouped_titles.apply(top)
```

```
[4]:
```

	Title	Language	2021-12	2022-01	2022-02
0	Jupyter Tutorial	de	30134.0	33295.0	19651.0
		en	6073.0	7716.0	6547.0
1	PyViz Tutorial	de	4873.0	3930.0	2573.0
		en	NaN	NaN	NaN
2	Python Basics	de	427.0	276.0	525.0
		en	95.0	226.0	157.0

What happened here? The upper function is called for each row group of the `DataFrame`, and then the results are concatenated with `pandas.concat`, labelling the parts with the group names. The result therefore has a hierarchical index whose inner level contains index values from the original `DataFrame`.

If you pass a function to `apply` that takes other arguments or keywords, you can pass them after the function:

```
[5]: grouped_titles = df.groupby("Title", as_index=False)
```

```
grouped_titles.apply(top, n=1)
```

```
[5]:
```

	Title	Language	2021-12	2022-01	2022-02
0	Jupyter Tutorial	de	30134.0	33295.0	19651.0
1	PyViz Tutorial	de	4873.0	3930.0	2573.0
2	Python Basics	de	427.0	276.0	525.0

We have now seen the basic usage of `apply`. What happens inside the passed function is very versatile and up to you; it only has to return a pandas object or a single value. In the following, we will therefore mainly show examples that can give you ideas on how to solve various problems with `groupby`.

First, let's look again at `describe`, called over the `GroupBy` object:

```
[6]: result = grouped_titles.describe()
```

```
result
```

```
[6]:
```

	2021-12				2022-01				2022-02					
	count	mean	std	min	25%	50%	75%	count	mean	std	min	25%	50%	75%
0	2.0	18103.5	17013.696262	6073.0	12088.25	18103.5	24118.75	2.0	20505.5	26900.25	33295.0	2.0	13099.0	19651.0
1	1.0	4873.0	NaN	4873.0	4873.00	4873.0	4873.00	1.0	3930.0	3930.00	3930.0	1.0	2573.0	2573.0
2	2.0	261.0	234.759451	95.0	178.00	261.0	344.00	2.0	251.0	260.215295	276.0	2.0	341.0	525.0

[3 rows x 24 columns]

When you call a method like `describe` within `GroupBy`, it is actually just an abbreviation for:

```
[7]: f = lambda x: x.describe()
grouped_titles.apply(f)
```

```
[7]:
```

	2021-12				2022-01				2022-02			
0	count	2.000000	2.000000	2.000000	count	2.000000	2.000000	2.000000	count	2.000000	2.000000	2.000000
	mean	18103.500000	20505.500000	13099.000000		20505.500000	26900.250000	33295.000000		13099.000000	19651.000000	19651.000000
	std	17013.696262	18087.084356	9265.927261		18087.084356	7716.000000	6547.000000		18087.084356	7716.000000	6547.000000
	min	6073.000000	7716.000000	6547.000000		6073.000000	7716.000000	6547.000000		6073.000000	7716.000000	6547.000000
	25%	12088.250000	14110.750000	9823.000000		12088.250000	14110.750000	9823.000000		12088.250000	14110.750000	9823.000000
	50%	18103.500000	20505.500000	13099.000000		18103.500000	20505.500000	13099.000000		18103.500000	20505.500000	13099.000000
	75%	24118.750000	26900.250000	16375.000000		24118.750000	26900.250000	16375.000000		24118.750000	26900.250000	16375.000000
	max	30134.000000	33295.000000	19651.000000		30134.000000	33295.000000	19651.000000		30134.000000	33295.000000	19651.000000
1	count	1.000000	1.000000	1.000000	count	1.000000	1.000000	1.000000	count	1.000000	1.000000	1.000000
	mean	4873.000000	3930.000000	2573.000000		4873.000000	3930.000000	2573.000000		4873.000000	3930.000000	2573.000000
	std	NaN	NaN	NaN		NaN	NaN	NaN		NaN	NaN	NaN
	min	4873.000000	3930.000000	2573.000000		4873.000000	3930.000000	2573.000000		4873.000000	3930.000000	2573.000000
	25%	4873.000000	3930.000000	2573.000000		4873.000000	3930.000000	2573.000000		4873.000000	3930.000000	2573.000000
	50%	4873.000000	3930.000000	2573.000000		4873.000000	3930.000000	2573.000000		4873.000000	3930.000000	2573.000000
	75%	4873.000000	3930.000000	2573.000000		4873.000000	3930.000000	2573.000000		4873.000000	3930.000000	2573.000000
	max	4873.000000	3930.000000	2573.000000		4873.000000	3930.000000	2573.000000		4873.000000	3930.000000	2573.000000
2	count	2.000000	2.000000	2.000000	count	2.000000	2.000000	2.000000	count	2.000000	2.000000	2.000000
	mean	261.000000	251.000000	341.000000		261.000000	251.000000	341.000000		261.000000	251.000000	341.000000
	std	234.759451	35.355339	260.215295		234.759451	35.355339	260.215295		234.759451	35.355339	260.215295
	min	95.000000	226.000000	157.000000		95.000000	226.000000	157.000000		95.000000	226.000000	157.000000
	25%	178.000000	238.500000	249.000000		178.000000	238.500000	249.000000		178.000000	238.500000	249.000000
	50%	261.000000	251.000000	341.000000		261.000000	251.000000	341.000000		261.000000	251.000000	341.000000
	75%	344.000000	263.500000	433.000000		344.000000	263.500000	433.000000		344.000000	263.500000	433.000000
	max	427.000000	276.000000	525.000000		427.000000	276.000000	525.000000		427.000000	276.000000	525.000000

Suppression of the group keys

In the previous examples, you saw that the resulting object has a hierarchical index formed by the group keys together with the indices of the individual parts of the original object. You can disable this by passing `group_keys=False` to `groupby`:

```
[8]: grouped_lang = df.groupby("Language", group_keys=False)

grouped_lang.apply(top)
```

```
[8]:
```

		2021-12	2022-01	2022-02
Title	Language			
Jupyter Tutorial	de	30134.0	33295.0	19651.0
PyViz Tutorial	de	4873.0	3930.0	2573.0
Python Basics	de	427.0	276.0	525.0
Jupyter Tutorial	en	6073.0	7716.0	6547.0
Python Basics	en	95.0	226.0	157.0
PyViz Tutorial	en	NaN	NaN	NaN

Quantile and bucket analysis

As described in *discretisation and grouping*, pandas has some tools, especially `cut` and `qcut`, to split data into buckets with bins of your choice or by sample quantiles. Combine these functions with `groupby` and you can conveniently perform bucket or quantile analysis on a dataset. Consider a simple random data set and a bucket categorisation of equal length with `cut`:

```
[9]: rng = np.random.default_rng()
df2 = pd.DataFrame(
    {
        "data1": rng.normal(size=1000),
        "data2": rng.normal(size=1000)
    }
)

quartiles = pd.cut(df2.data1, 4)

quartiles[:10]
```

```
[9]: 0    (-1.38, 0.0424]
1    (-1.38, 0.0424]
2    (0.0424, 1.464]
3    (0.0424, 1.464]
4    (-1.38, 0.0424]
5    (0.0424, 1.464]
6    (-1.38, 0.0424]
7    (-1.38, 0.0424]
8    (-1.38, 0.0424]
9    (-1.38, 0.0424]
Name: data1, dtype: category
Categories (4, interval[float64, right]): [(-2.807, -1.38] < (-1.38, 0.0424] < (0.0424, 1.464] < (1.464, 2.886]]
```

The category object returned by `cut` can be passed directly to `groupby`. So we could calculate a set of group statistics for the quartiles as follows:

```
[10]: def stats(group):
      return pd.DataFrame(
          {
              "min": group.min(),
              "max": group.max(),
              "count": group.count(),
              "mean": group.mean(),
          }
      )

grouped_quart = df2.groupby(quartiles)

grouped_quart.apply(stats)
```

```
[10]:
```

		min	max	count	mean
data1					
(-2.807, -1.38]	data1	-2.801488	-1.389215	74	-1.791025
	data2	-2.435057	2.248734	74	-0.052099
(-1.38, 0.0424]	data1	-1.369729	0.042258	452	-0.557848
	data2	-2.815141	2.555586	452	0.007063
(0.0424, 1.464]	data1	0.043187	1.453391	401	0.661546
	data2	-2.680655	3.002200	401	0.055687
(1.464, 2.886]	data1	1.471820	2.886283	73	1.832881
	data2	-2.108169	3.566785	73	0.145811

These were buckets of equal length; to calculate buckets of equal size based on sample quantiles, we can use `qcut`. I pass `labels=False` to get only quantile numbers:

```
[11]: quartiles_samp = pd.qcut(df2.data1, 4, labels=False)

grouped_quart_samp = df2.groupby(quartiles_samp)

grouped_quart_samp.apply(stats)
```

```
[11]:
```

		min	max	count	mean
data1					
0	data1	-2.801488	-0.651380	250	-1.207350
	data2	-2.815141	2.555586	250	-0.048382
1	data1	-0.648977	-0.027545	250	-0.332227
	data2	-2.529867	2.478078	250	0.069973
2	data1	-0.026572	0.691897	250	0.325120
	data2	-2.638173	3.002200	250	0.063670
3	data1	0.692362	2.886283	250	1.272046
	data2	-2.680655	3.566785	250	0.043986

Populating data with group-specific values

When cleaning missing data, in some cases you will replace data observations with `dropna`, but in other cases you may want to fill the null values (NA) with a fixed value or a value derived from the data. `fillna` is the right tool for this; here, for example, I fill the null values with the mean:

```
[12]: s = pd.Series(rng.normal(size=8))
      s[::3] = np.nan
```

s

```
[12]: 0      NaN
      1    0.835698
      2   -0.262870
      3      NaN
      4   -1.345111
      5   -0.266797
      6      NaN
      7    0.550379
      dtype: float64
```

```
[13]: s.fillna(s.mean())
```

```
[13]: 0   -0.097740
      1    0.835698
      2   -0.262870
      3   -0.097740
      4   -1.345111
      5   -0.266797
      6   -0.097740
      7    0.550379
      dtype: float64
```

Here are some sample data for my tutorials, divided into German and English editions:

Suppose you want the fill value to vary by group. These values can be predefined, and since the groups have an internal name attribute, you can use this with `apply`:

```
[14]: fill_values = {"de": 10632, "en": 3469}

      fill_func = lambda g: g.fillna(fill_values[g.name])

      df.groupby("Language").apply(fill_func)
```

```
[14]:
```

	Language	Title	Language	2021-12	2022-01	2022-02
de	Jupyter Tutorial	de	30134.0	33295.0	19651.0	
	PyViz Tutorial	de	4873.0	3930.0	2573.0	
	Python Basics	de	427.0	276.0	525.0	
en	Jupyter Tutorial	en	6073.0	7716.0	6547.0	
	PyViz Tutorial	en	3469.0	3469.0	3469.0	
	Python Basics	en	95.0	226.0	157.0	

You can also group the data and use `apply` with a function that calls `fillna` for each data packet:

```
[15]: fill_mean = lambda g: g.fillna(g.mean())

df.groupby("Language").apply(fill_mean)
```

			2021-12	2022-01	2022-02
Language	Title	Language			
de	Jupyter Tutorial	de	30134.0	33295.0	19651.0
	PyViz Tutorial	de	4873.0	3930.0	2573.0
	Python Basics	de	427.0	276.0	525.0
en	Jupyter Tutorial	en	6073.0	7716.0	6547.0
	PyViz Tutorial	en	3084.0	3971.0	3352.0
	Python Basics	en	95.0	226.0	157.0

Group weighted average

Since operations between columns in a DataFrame or two Series are possible, we can calculate the group-weighted average, for example:

```
[16]: df3 = pd.DataFrame(
    {
        "category": ["de", "de", "de", "de", "en", "en", "en", "en"],
        "data": np.random.randint(100000, size=8),
        "weights": np.random.rand(8),
    }
)

df3
```

	category	data	weights
0	de	41970	0.967458
1	de	53639	0.605162
2	de	16329	0.007546
3	de	14668	0.033338
4	en	99258	0.826135
5	en	7727	0.861027
6	en	13388	0.005460
7	en	27957	0.276577

The group average weighted by category would then be:

```
[17]: grouped_cat = df3.groupby("category")
get_wavg = lambda g: np.average(g["data"], weights=g["weights"])

grouped_cat.apply(get_wavg)
```

	category
de	45662.558991
en	48983.872414

dtype: float64

Correlation

An interesting task could be to calculate a `DataFrame` consisting of the percentage changes.

For this purpose, we first create a function that calculates the pairwise correlation of the 2021-12 column with the subsequent columns:

```
[18]: corr = lambda x: x.corrwith(x["2021-12"])
```

Next, we calculate the percentage change:

```
[19]: pcts = df.pct_change().dropna()
```

Finally, we group these percentage changes by year, which can be extracted from each row label with a one-line function that returns the year attribute of each date label:

```
[20]: grouped_lang = pcts.groupby("Language")
grouped_lang.apply(corr)
```

```
[20]:
```

	2021-12	2022-01	2022-02
Language			
de	1.0	1.000000	1.000000
en	1.0	0.699088	0.99781

```
[21]: grouped_lang.apply(lambda g: g["2021-12"].corr(g["2022-01"]))
```

```
[21]: Language
de    1.000000
en    0.699088
dtype: float64
```

Performance problems with apply

Since the `apply` method typically acts on each individual value in a `Series`, the function is called once for each value. If you have thousands of values, the function will be called thousands of times. This ignores the fast vectorisations of pandas unless you are using NumPy functions and slow Python is used. For example, we previously grouped the data by title and then called our `top` method with `apply`. Let's measure the time for this:

```
[22]: %%timeit
grouped_titles.apply(top)

566 µs ± 8.04 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

We can get the same result without applying by passing the `DataFrame` to our `top` method:

```
[23]: %%timeit
top(df)

43.8 µs ± 693 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

This calculation is 18 times faster.

Optimising apply with Cython

It is not always easy to find an alternative for `apply`. However, numerical operations like our `top` method can be made faster with [Cython](#). To use Cython in Jupyter, we use the following *IPython magic*:

```
[24]: %load_ext Cython
```

Then we can define our `top` function with Cython:

```
[25]: %%cython
def top_cy(df, n=5, column="2021-12"):
    return df.sort_values(by=column, ascending=False)[:n]
```

```
[26]: %%timeit
grouped_titles.apply(top_cy)

565 µs ± 7.08 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

We haven't really gained much with this yet. Further optimisation possibilities would be to define the type in the Cython code with `cpdef`. For this, however, we would have to modify our method, because then no `DataFrame` can be passed.

2.4.16 Pivot tables and crosstabs

A [pivot table](#) is a data summary tool often found in spreadsheet and other data analysis software. It summarises a table of data by one or more keys and arranges the data in a rectangle, with some of the group keys along the rows and some along the columns. Pivot tables in Python with pandas are made possible by the [groupby](#) function in combination with reshaping operations using [hierarchical indexing](#). `DataFrame` has a `pivot_table` method, and there is also a top-level function `pandas.pivot_table`. `pivot_table` not only provides a convenient interface to `groupby`, but can also add partial sums (margins).

Suppose we wanted to compute a table of group averages (the default aggregation type of `pivot_table`) ordered by title and language in the rows:

```
[1]: import numpy as np
import pandas as pd
```

```
[2]: df = pd.DataFrame(
    {
        "Title": [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            "PyViz Tutorial",
            "Python Basics",
            "Python Basics",
        ],
        "Language": ["de", "en", "de", None, "de", "en"],
        "2021-12": [30134, 6073, 4873, None, 427, 95],
        "2022-01": [33295, 7716, 3930, None, 276, 226],
        "2022-02": [19651, 6547, 2573, None, 525, 157],
    }
)

df
```

```
[2]:
```

	Title	Language	2021-12	2022-01	2022-02
0	Jupyter Tutorial	de	30134.0	33295.0	19651.0
1	Jupyter Tutorial	en	6073.0	7716.0	6547.0
2	PyViz Tutorial	de	4873.0	3930.0	2573.0
3	PyViz Tutorial	None	NaN	NaN	NaN
4	Python Basics	de	427.0	276.0	525.0
5	Python Basics	en	95.0	226.0	157.0

```
[3]: df.pivot_table(index=["Title", "Language"])
```

```
[3]:
```

Title	Language	2021-12	2022-01	2022-02
Jupyter Tutorial	de	30134.0	33295.0	19651.0
	en	6073.0	7716.0	6547.0
PyViz Tutorial	de	4873.0	3930.0	2573.0
Python Basics	de	427.0	276.0	525.0
	en	95.0	226.0	157.0

This could also have been done directly with `groupby`.

Now let's say we want to get the mean of hits of all languages per title for each individual month. For this I will enter Title in the table columns and the months in the rows:

```
[4]: df.pivot_table(columns="Title")
```

```
[4]:
```

Title	Jupyter Tutorial	PyViz Tutorial	Python Basics
2021-12	18103.5	4873.0	261.0
2022-01	20505.5	3930.0	251.0
2022-02	13099.0	2573.0	341.0

Alternatively, we can keep the languages as columns and add the mean values by specifying `margins=True`:

```
[5]: df.pivot_table(columns=["Title", "Language"], margins=True)
```

```
[5]:
```

Title	Jupyter Tutorial			PyViz Tutorial			\
	de	en	All	de	All		
2021-12	30134.0	6073.0	18103.5	4873.0	4873.0		
2022-01	33295.0	7716.0	20505.5	3930.0	3930.0		
2022-02	19651.0	6547.0	13099.0	2573.0	2573.0		
Title	Python Basics						
	de	en	All				
2021-12	427.0	95.0	261.0				
2022-01	276.0	226.0	251.0				
2022-02	525.0	157.0	341.0				

To use an aggregation function other than `mean`, pass it to the keyword argument `aggfunc`. With `sum`, for example, you get the sum:

```
[6]: df.pivot_table(columns=["Title", "Language"], aggfunc=sum, margins=True)
```

```
[6]:
```

Title	Jupyter Tutorial			PyViz Tutorial			\
	de	en	All	de	All		
2021-12	30134.0	6073.0	36207.0	4873.0	4873.0		
2022-01	33295.0	7716.0	41011.0	3930.0	3930.0		

(continues on next page)

(continued from previous page)

```

2022-02          19651.0  6547.0  26198.0          2573.0  2573.0

Title    Python Basics
Language      de      en    All
2021-12      427.0   95.0  522.0
2022-01      276.0  226.0  502.0
2022-02      525.0  157.0  682.0

```

`pivot_table` options:

Function name	Description
<code>values</code>	column name(s) to aggregate; by default, all numeric columns are aggregated
<code>index</code>	column names or other group keys to be grouped in the rows of the resulting pivot table
<code>columns</code>	column names or other group keys to be grouped in the columns of the resulting pivot table
<code>aggfunc</code>	aggregation function or list of functions (by default <code>mean</code>); can be any function valid in a <code>groupby</code> context
<code>fill_value</code>	replaces missing values in the result table
<code>dropna</code>	if <code>True</code> , columns whose entries are all <code>NA</code> are ignored
<code>margins</code>	inserts row/column subtotals and grand totals (default: <code>False</code>)
<code>margins_name</code>	name used for row/column labels if <code>margins=True</code> is passed, default is <code>All</code> .
<code>observed</code>	For categorical group keys, if <code>True</code> , only the observed category values are displayed in the keys and not all categories

Crosstabs

A crosstab is a special case of a pivot table that calculates the frequency of groups. For example, in the context of an analysis of this data, we might want to determine which title was published in which language, so we could use `pivot_table` for this, but the function `pandas.crosstab` is more convenient.

```
[7]: pd.crosstab(df.Title, df.Language)
```

```

[7]: Language      de  en
Title
Jupyter Tutorial    1   1
PyViz Tutorial      1   0
Python Basics      1   1

```

The first two arguments for `crosstab` can each be either an array or a series or a list of arrays.

With `margins=True` we can also calculate the sums of the columns and rows as well as the total sum:

```
[8]: pd.crosstab(df.Title, df.Language, margins=True)
```

```

[8]: Language      de  en  All
Title
Jupyter Tutorial    1   1    2
PyViz Tutorial      1   0    1
Python Basics      1   1    2
All                  3   2    5

```

2.4.17 Convert dtype

Sometimes the pandas data types do not fit really well. This can be due to serialisation formats that do not contain type information, for example. However, sometimes you should also change the type to achieve better performance – either more manipulation possibilities or less memory requirements. In the following examples, we will make different conversions of a `Series`:

```
[1]: import numpy as np
import pandas as pd
```

```
[2]: rng = np.random.default_rng()
s = pd.Series(rng.normal(size=7))
```

```
[3]: s
[3]: 0    0.330852
1    0.703606
2   -0.773463
3   -0.814739
4    1.677859
5    0.992312
6    0.175644
dtype: float64
```

Automatic conversion

`pandas.Series.convert_dtypes` tries to convert a `Series` to a type that supports NA. In the case of our `Series`, the type is changed from `float64` to `Float64`:

```
[4]: s.convert_dtypes()
```

```
[4]: 0    0.330852
1    0.703606
2   -0.773463
3   -0.814739
4    1.677859
5    0.992312
6    0.175644
dtype: Float64
```

Unfortunately, however, with `convert_dtypes` I have little control over what data type is converted to. Therefore, I prefer `pandas.Series.astype`:

```
[5]: s.astype("Float32")
```

```
[5]: 0    0.330852
1    0.703606
2   -0.773463
3   -0.814739
4    1.677859
5    0.992312
6    0.175644
dtype: Float32
```

Using the correct type can save memory. The usual data type is 8 bytes wide, for example `int64` or `float64`. If you can use a narrower type, this will significantly reduce memory consumption, allowing you to process more data. You can use NumPy to check the limits of integer and float types:

```
[6]: np.iinfo("int64")
[6]: iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)

[7]: np.finfo("float32")
[7]: finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38, dtype=float32)

[8]: np.finfo("float64")
[8]: finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308,
↳ dtype=float64)
```

Memory usage

To calculate the memory consumption of the `Series`, you can use `pandas.Series.nbytes` to determine the memory used by the data. `pandas.Series.memory_usage` also records the index memory and the data type. With `deep=True` you can also determine the memory consumption at system level.

```
[9]: s.nbytes
[9]: 56

[10]: s.astype("Float32").nbytes
[10]: 35

[11]: s.memory_usage()
[11]: 188

[12]: s.astype("Float32").memory_usage()
[12]: 167

[13]: s.memory_usage(deep=True)
[13]: 188
```

String and category types

The `pandas.Series.astype` method can also convert numeric series into strings if you pass `str`. Note the `dtype` in the following example:

```
[14]: s.astype(str)
[14]: 0    0.33085233447486595
      1    0.7036061214691522
      2   -0.7734631836438829
      3   -0.8147390382513203
```

(continues on next page)

(continued from previous page)

```

4      1.6778586038914356
5      0.9923123929031976
6      0.17564372049973478
dtype: object

```

```
[15]: s.astype(str).memory_usage()
```

```
[15]: 188
```

```
[16]: s.astype(str).memory_usage(deep=True)
```

```
[16]: 661
```

To convert to a categorical type, you can pass 'category' as the type:

```
[17]: s.astype(str).astype("category")
```

```

[17]: 0      0.33085233447486595
      1      0.7036061214691522
      2     -0.7734631836438829
      3     -0.8147390382513203
      4      1.6778586038914356
      5      0.9923123929031976
      6      0.17564372049973478
dtype: category
Categories (7, object): ['-0.7734631836438829', '-0.8147390382513203', '0.
↪ 17564372049973478', '0.33085233447486595', '0.7036061214691522', '0.9923123929031976',
↪ '1.6778586038914356']

```

A categorical Series is useful for string data and can lead to large memory savings. This is because when converting to categorical data, pandas no longer uses Python strings for each value, but repeating values are not duplicated. You still have all the features of the `str` attribute, but you save a lot of memory when you have a lot of duplicate values and you increase performance because you don't have to do as many string operations.

```
[18]: s.astype("category").memory_usage(deep=True)
```

```
[18]: 495
```

Ordered categories

To create ordered categories, you need to define your own `pandas.CategoricalDtype`:

```
[19]: from pandas.api.types import CategoricalDtype
```

```

sorted = pd.Series(sorted(set(s)))
cat_dtype = CategoricalDtype(categories=sorted, ordered=True)

s.astype(cat_dtype)

```

```

[19]: 0      0.330852
      1      0.703606
      2     -0.773463

```

(continues on next page)

(continued from previous page)

```

3  -0.814739
4   1.677859
5   0.992312
6   0.175644
dtype: category
Categories (7, float64): [-0.814739 < -0.773463 < 0.175644 < 0.330852 < 0.703606 < 0.
↪ 992312 < 1.677859]

```

```
[20]: s.astype(cat_dtype).memory_usage(deep=True)
```

```
[20]: 495
```

The following table lists the types you can pass to `astype`.

Data type	Description
<code>str, 'str'</code>	convert to Python string
<code>'string'</code>	convert to Pandas string with <code>pandas.NA</code>
<code>int, 'int', 'int64'</code>	convert to NumPy <code>int64</code>
<code>'int32', 'uint32'</code>	convert to NumPy <code>int32</code>
<code>'Int64'</code>	convert to pandas <code>Int64</code> with <code>pandas.NA</code>
<code>float, 'float', 'float64'</code>	convert to floats
<code>'category'</code>	convert to <code>CategoricalDtype</code> with <code>pandas.NA</code>

Conversion to other data types

The `pandas.Series.to_numpy` method or the `pandas.Series.values` property gives us a NumPy array of values, and `pandas.Series.to_list` returns a Python list of values. Why would you want to do this? pandas objects are usually much more user-friendly and the code is easier to read. Also, python lists will be much slower to process. With `pandas.Series.to_frame` you can create a DataFrame with a single column, if necessary:

```
[21]: s.to_frame()
```

```

[21]:      0
0  0.330852
1  0.703606
2 -0.773463
3 -0.814739
4  1.677859
5  0.992312
6  0.175644

```

The function `pandas.to_datetime` can also be useful to convert values in pandas to date and time.

READ, PERSIST AND PROVIDE DATA

You can get an overview of public repositories with research data e.g. in *Open data*.

In addition to specific Python libraries for accessing *Overview* and *Geodata*, we will introduce you to different *serialisation formats* and three tools in more detail that make data accessible:

- *pandas IO tools*
- *htpx*
- *Intake*

See also:

pandas I/O API

The pandas I/O API is a set of top level **reader** functions that return a pandas object. In most cases corresponding **write** methods are also available.

Scrapy

Framework for extracting data from websites as JSON, CSV or XML files.

Pattern

Python module for data mining, natural language processing, ML and network analysis.

Web Scraping Reference

Overview of web scraping with Python.

We introduce *PostgreSQL*, *SQLAlchemy* and *PostGIS* for storing relational data, Python objects and geodata.

For the storage of other data types we introduce you to different *NoSQL databases and concepts*.

Next, we will show you how to provide the data via an *Application Programming Interface (API)*.

With *DVC* we present you a tool that allows data provenance, i.e. the traceability of the origin of the data and the way they are created.

Finally in the next chapter you will learn some good practices and helpful Python packages to *clean up and validate data*.

3.1 Open data

You can get an overview of public repositories with research data e.g. in

- [Registry of research data repositories \(re3data\)](#)
- [Awesome Public Datasets](#)
- [Public APIs](#)
- [Machine learning datasets](#)
- [Roboflow Computer Vision Datasets](#)
- [DBpedia](#)
- [World Health Data Platform/Global Health Observatory](#)
- [UNICEF Data](#)
- [IATI Registry](#)
- [World Bank Open Data](#)
- [Open Data Inception](#)
- [European data](#)
- [GovData.de](#)
- [US Census Bureau](#)
- [data.gov](#)
- [Google Dataset Search](#)
- [Google Public Data Search](#)
- [Registry of Open Data on AWS](#)
- [Yelp Open Dataset](#)
- [Kaggle Datasets](#)
- [OpenDataMonitor](#)
- [Open Data Impact Map](#)
- [CKAN](#)
- [UC Irvine Machine Learning Repository](#)
- [Hugging Face Datasets](#)
- [Dataverse Project](#)
- [Open Data Kit](#)
- [LODUM University of Münster's Open Data initiative](#)
- [freeCodeCamp open-data](#)
- [Reddit Datasets Community](#)

3.2 pandas IO tools

pandas has a number of functions for reading table data as DataFrame objects, including

Function	Description
<code>pan-das.read_csv</code>	loads <i>CSV</i> data from a file, URL or file-like object; usually a comma is used as separator
<code>pan-das.read_fwf</code>	loads FWF (fixed-width files), which is data in column format with a fixed width
<code>pan-das.read_clipboard</code>	reads data from the clipboard and passes it to <code>read_csv</code> ; useful for converting tables from web pages, among other things
<code>pan-das.read_excel</code>	reads table data from an <i>Excel</i> XLS or XLSX file
<code>pan-das.read_hdf</code>	reads HDF5 (Hierarchical Data Format) files
<code>pan-das.read_html</code>	reads all tables from the specified <i>HTML</i> document
<code>pan-das.read_json</code>	reads data from a <i>JSON</i> file
<code>pan-das.read_feather</code>	reads the <i>Feather</i> binary file format
<code>pan-das.read_orc</code>	reads Apache ORC (Optimized Row Columnar) binary data
<code>pan-das.read_parquet</code>	reads <i>Apache Parquet</i> binary file format
<code>pan-das.read_pickle</code>	reads any object stored in Python <i>Pickle</i> format
<code>pan-das.read_sas</code>	reads a SAS (Statistical Analysis System) data set
<code>pan-das.read_spss</code>	reads a data file created by <i>SPSS</i>
<code>pan-das.read_sql</code>	reads the results of an SQL query (with <i>SQLAlchemy</i>) as a pandas DataFrame
<code>pan-das.read_sql_table</code>	reads an entire SQL table (with <i>SQLAlchemy</i>) as a pandas DataFrame (corresponds to a query that selects everything in this table with <code>read_sql</code>)
<code>pan-das.read_stata</code>	reads a data set from the <i>Stata</i> file format

See also:

pandas I/O API

The pandas I/O API is a collection of **reader** functions that return a pandas object. In most cases, corresponding **writer** methods are also available.

First, I will give an overview of some of these functions that are designed to convert text and excel data into a pandas DataFrame: *CSV*, *JSON* and *Excel*. The optional arguments for these functions can be divided into the following categories:

Indexing

Can one or more columns index the returned DataFrame, and whether the column names should be retrieved from the file, the arguments you specify, or not at all.

Type inference and data conversion

This includes the custom value conversions and the custom list of missing value flags.

Date and time parsing

This includes the combining capability, including combining date and time information spread across multiple columns into a single column in the result.

Iteration

Support for iteration over parts of very large files.

Problems with unclean data

Skipping of rows or footers, comments or other trivia such as numeric data with thousands separated by commas.

Since data can be very messy in the real world, some of the data loading functions (especially `read_csv`) have accumulated a long list of optional arguments over time. The online documentation for pandas contains many examples of each function.

Some of these functions, like `pandas.read_csv`, perform type inference because the data types of the columns are not part of the data format. This means that you don't necessarily have to specify which columns are numeric, integer, boolean or string. With other data formats such as HDF5, ORC and Parquet, however, the data type information is already embedded in the format.

3.3 Serialisation formats

Data serialisation converts structured data to a format that allows sharing and or storing of the data. Before serialising data you have to decide how the data should be structured – flat or nested. The differences in the two styles are shown in the examples below:

Flat JSON style:

```
{
  "id"       : "veit",
  "first_name" : "Veit",
  "last_name"  : "Schiele",
}
```

Nested JSON style:

```
{
  "veit" : {
    "first_name" : "Veit",
    "last_name"  : "Schiele",
  },
}
```

3.3.1 Data serialisation

If the data is to be serialised flat, Python offers two functions:

repr

`repr()` outputs a printable representation of the input, for example:

```
[1]: data = { "id" : "veit", "first_name": "Veit", "last_name": "Schiele" }

print(repr(data))

{'id': 'veit', 'first_name': 'Veit', 'last_name': 'Schiele'}
```

```
[2]: with open("data.txt", "w") as f:
      f.write(repr(data))
```

ast.literal_eval

The `ast.literal_eval()` function parses and analyses the Python data type of an expression. Supported data types are strings, numbers, tuples, lists, dictionaries and `None`.

```
[3]: import ast

with open("data.txt", "r") as f:
    d = ast.literal_eval(f.read())

print(d)

{'id': 'veit', 'first_name': 'Veit', 'last_name': 'Schiele'}
```

3.3.2 CSV

Overview

Data structure support	--	CSV is used to store tabular data, but unlike other serialisation formats reviewed here, it's not suitable for (nested) objects.
Standardisation	--	CSV is not well standardised: neither the encoding is defined nor the separation of the cell contents (comma, semicolon etc.).
Schem: IDL	--	No
Language support	++	The CSV format is well supported by almost every programming language. A <code>csv</code> module is included in the Python standard library and <code>pandas</code> can read a CSV file straight into a <code>Dataframe</code> . Even if CSV is the only format described here that is well supported by spreadsheet programs like Excel, you should see if you can import more structured Excel files directly, e.g. with <code>pandas read_excel</code> .
Human readability	+-	CSV is readable especially for integer or decimal numbers with the same character length. In all other cases it will be difficult to identify the corresponding columns.
Speed	+	CSV is very fast to serialise and deserialise.
File size	++	Only <i>Protocol Buffers (Protobuf)</i> should be more compact.

See also:

- [RFC 4180](#)
- `xsv`

Example

iris.csv

```
5.1,0.22222222,3.5,0.625,1.4,0.06779661,0.2,0.04166667,setosa
4.9,0.16666667,3,0.41666667,1.4,0.06779661,0.2,0.04166667,setosa
4.7,0.11111111,3.2,0.5,1.3,0.050847458,0.2,0.04166667,setosa
4.6,0.08333333,3.1,0.45833333,1.5,0.084745763,0.2,0.04166667,setosa
5,0.19444444,3.6,0.66666667,1.4,0.06779661,0.2,0.04166667,setosa
...
```

CSV example

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

After importing pandas, we first read a csv file with `read_csv`:

```
[2]: pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
    ↪data/books.csv"
)
```

```
[2]:
```

	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
0	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
1	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
2	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

As you can see, this file has no header. To give the DataFrame a header, you have several options. You can allow pandas to assign default column names, or you can define the names yourself:

```
[3]: pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
    ↪data/books.csv",
    header=None,
)
```

```
[3]:
```

	0	1	2	3	4
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

```
[4]: pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
    ↪data/books.csv",
    names=["Title", "Language", "Authors", "License", "Publication date"],
)
```

```
[4]:
```

	Title	Language	Authors	License	Publication date
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

Suppose you want the Authors column to be the index of the returned DataFrame. You can either specify that you want the column at index 3 or with the name Authors by using the argument `index_col`:

```
[5]: pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
    ↪data/books.csv",
    index_col=["Authors"],
    names=["Title", "Language", "Authors", "License", "Publication date"],
)
```

```
[5]:
```

	Title	Language	License	Publication date
Authors				

(continues on next page)

(continued from previous page)

Veit Schiele	Python basics	en	BSD-3-Clause	2021-10-28
Veit Schiele	Jupyter Tutorial	en	BSD-3-Clause	2019-06-27
Veit Schiele	Jupyter Tutorial	de	BSD-3-Clause	2020-10-26
Veit Schiele	PyViz Tutorial	en	BSD-3-Clause	2020-04-13

In case you want to build a hierarchical index from several columns, pass a list of column numbers or names:

```
[6]: pd.read_csv(
      "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
      ↪data/books.csv",
      index_col=[2, 0],
      names=["Title", "Language", "Authors", "License", "Publication date"],
    )
```

```
[6]:
```

		Language	License	Publication date
Authors	Title			
Veit Schiele	Python basics	en	BSD-3-Clause	2021-10-28
	Jupyter Tutorial	en	BSD-3-Clause	2019-06-27
	Jupyter Tutorial	de	BSD-3-Clause	2020-10-26
	PyViz Tutorial	en	BSD-3-Clause	2020-04-13

In some cases, a table does not have a fixed separator, but uses several spaces or some other pattern to separate fields. Suppose a file looks like this:

```
[7]: list(open("books.txt"))
```

```
[7]: [' Title           Language Authors      License      Publication date\n',
      '1 Python basics    en      Veit Schiele BSD-3-Clause 2021-10-28\n',
      '2 Jupyter Tutorial  en      Veit Schiele BSD-3-Clause 2019-06-27\n',
      '3 Jupyter Tutorial  de      Veit Schiele BSD-3-Clause 2020-10-26\n',
      '4 PyViz Tutorial    en      Veit Schiele BSD-3-Clause 2020-04-13\n']
```

In such cases, you can pass a regular expression as a separator for `read_csv`. This can be expressed by the regular expression `\s\s+`, so then we have:

```
[8]: pd.read_csv("books.txt", sep="\s\s+", engine="python")
```

```
[8]:
```

	Title	Language	Authors	License	Publication date
1	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
2	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
3	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
4	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

Since there was one column name less than the number of data rows, `read_csv` infers that in this case the first column should be the index of the DataFrame.

The parser functions have many additional arguments that help you handle the wide variety of exception file formats that occur. For example, you can skip individual lines of a file with `skiprows`:

```
[9]: pd.read_csv(
      "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
      ↪data/books.csv",
      skiprows=[2],
      names=["Title", "Language", "Authors", "License", "Publication date"],
    )
```



```
[9]:
```

	Title	Language	Authors	License	Publication date
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

Dealing with missing values is an important and often complicated part of parsing data. Missing data is usually either not present (empty string) or indicated by a placeholder. By default, pandas uses a number of common placeholders, such as NA and NULL:

```
[10]: df = pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
    ↪data/books.csv",
    names=[
        "Title",
        "Language",
        "Authors",
        "License",
        "Publication date",
        "doi",
    ],
)

df
```

```
[10]:
```

	Title	Language	Authors	License	Publication date	doi
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28	NaN
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27	NaN
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26	NaN
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13	NaN

```
[11]: df.isna()
```

```
[11]:
```

	Title	Language	Authors	License	Publication date	doi
0	False	False	False	False	False	True
1	False	False	False	False	False	True
2	False	False	False	False	False	True
3	False	False	False	False	False	True

The `na_values` option can take either a list or a series of strings to account for missing values:

```
[12]: pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
    ↪data/books.csv",
    na_values=["BSD-3-Clause"],
    names=[
        "Title",
        "Language",
        "Authors",
        "License",
        "Publication date",
        "doi",
    ],
)
```

```
[12]:
```

	Title	Language	Authors	License	Publication date	doi
0	Python basics	en	Veit Schiele	NaN	2021-10-28	NaN
1	Jupyter Tutorial	en	Veit Schiele	NaN	2019-06-27	NaN
2	Jupyter Tutorial	de	Veit Schiele	NaN	2020-10-26	NaN
3	PyViz Tutorial	en	Veit Schiele	NaN	2020-04-13	NaN

The most frequent arguments of the function `read_csv`:

Argument	Description
<code>path</code>	String specifying the location in the file system, a URL or a file-like object
<code>sep</code>	String or regular expression to separate the fields in each row
or <code>delim</code>	
<code>header</code>	Row number to be used as column name; default is 0, i.e. the first row, but should be None if there is no header row
<code>index_</code>	Row numbers or names to be used as row index in the result; can be a single name/number or a list of them for a hierarchical index
<code>names</code>	List of column names
<code>skipro</code>	Number of rows to be ignored at the beginning of the file or list of row numbers starting at 0 to be skipped
<code>na_val</code>	sequence of values to be replaced by NA
<code>commen</code>	character to separate comments from the end of the line
<code>parse_</code>	Attempt to parse data with datetime; defaults to False. If True, attempts to parse all columns. Otherwise, a list of column numbers or names to parse can be specified. If the list element is a tuple or a list, multiple columns are combined and converted to a date, for example if the date and time are split between two columns
<code>keep_d</code>	if columns are combined to parse the date, the combined columns are kept; default: False
<code>conver</code>	Dict containing the column number of names mapped to functions, for example {'Title': f} would apply the function f to all values in the column Title
<code>dayfir</code>	treat as an international format when parsing potentially ambiguous dates, for example 28/6/2021 → 28. Juni 2021; False by default
<code>date_p</code>	function to use for parsing dates
<code>nrows</code>	Number of lines to read from the beginning of the file.
<code>iterat</code>	Return a <code>TextFileReader</code> object to read the file piece by piece; this object can also be used with the <code>with</code> statement
<code>chunks</code>	For the iteration, the size of the data blocks.
<code>skip_f</code>	number of lines to be ignored at the end of the file
<code>verbo</code>	outputs various information about the parser output, for example the number of missing values in non-numeric columns
<code>encodi</code>	Text encoding for Unicode, for example <code>utf-8</code> for UTF-8 encoded text
<code>squeez</code>	if the parsed data contains only one column, a Series is returned
<code>thousa</code>	Separator for thousands, for example <code>,</code> or <code>.</code>

Reading in text files piece by piece

If you want to process very large files, you can also read in only a small part of a file or iterate through smaller parts of a file.

Before we look at a large file, we reduce the number of lines displayed with `options.display.max_rows`:

```
[13]: pd.options.display.max_rows = 10
```

```
[14]: pd.read_csv("example.csv")
```

```
[14]:
```

	Date	Mon.	Tues.	Wed.	Thurs.	Fri.	Sat.	\
0	1996-01-01	0.129453	-0.023836	1.121460	1.698286	-0.598506	1.042221	
1	1996-01-02	-0.094021	-0.727942	0.698641	-1.198040	1.927505	1.147445	
2	1996-01-03	-0.560857	0.145222	-0.990202	1.200214	0.717339	1.117095	
3	1996-01-04	-0.169755	-0.677391	-1.533519	-0.343477	-0.109705	1.038236	
4	1996-01-05	1.344705	-1.817261	0.460991	-0.839633	0.265814	0.477659	
...	
9127	2020-12-27	-0.881800	-0.074270	-0.351769	1.381641	-0.049548	1.664180	
9128	2020-12-28	-0.143386	0.198217	-1.243861	1.196576	1.338166	-0.212333	
9129	2020-12-29	0.398787	-0.848786	1.791707	-1.167592	-0.033881	-0.285559	
9130	2020-12-30	0.587846	0.411580	1.150380	0.444638	-1.093577	0.605456	
9131	2020-12-31	0.736350	0.436292	-0.260171	-0.066066	-0.328324	-0.586792	
	Sun.							
0		-0.726412						
1		-1.134103						
2		-1.793565						
3		-0.799088						
4		0.636383						
...	...							
9127		-1.032204						
9128		-0.023131						
9129		-0.323477						
9130		1.463345						
9131		-1.204582						

[9132 rows x 8 columns]

If you only want to read a small number of lines (without reading the whole file), you can specify this with `nrows`:

```
[15]: pd.read_csv("example.csv", nrows=7)
```

```
[15]:
```

	Date	Mon.	Tues.	Wed.	Thurs.	Fri.	Sat.	\
0	1996-01-01	0.129453	-0.023836	1.121460	1.698286	-0.598506	1.042221	
1	1996-01-02	-0.094021	-0.727942	0.698641	-1.198040	1.927505	1.147445	
2	1996-01-03	-0.560857	0.145222	-0.990202	1.200214	0.717339	1.117095	
3	1996-01-04	-0.169755	-0.677391	-1.533519	-0.343477	-0.109705	1.038236	
4	1996-01-05	1.344705	-1.817261	0.460991	-0.839633	0.265814	0.477659	
5	1996-01-06	-0.354445	-0.065182	-1.244963	-0.559732	0.042362	-0.303712	
6	1996-01-07	1.460922	0.164412	0.883960	-0.833642	0.001582	1.138469	
	Sun.							
0		-0.726412						

(continues on next page)

(continued from previous page)

```

1 -1.134103
2 -1.793565
3 -0.799088
4  0.636383
5  0.067632
6  0.561618

```

To read a file piece by piece, you can specify the number of lines with `chunksize`:

```

[16]: pd.read_csv("example.csv", chunksize=1000)
[16]: <pandas.io.parsers.readers.TextFileReader at 0x11fa09110>

```

The `TextFileReader` object returned by `read_csv` allows iteration over parts of the file according to the `chunksize`. For example, we can iterate over the `example.csv` file and aggregate the number of values in the `Date` column as follows:

```

[17]: chunks = pd.read_csv("example.csv", chunksize=1000)

serie = pd.Series([], dtype="float64")
for chunk in chunks:
    values = serie.add(chunk["Date"].value_counts(), fill_value=0)

sorted_values = values.sort_values(ascending=False)

```

```

[18]: sorted_values[:10]

```

```

[18]: Date
2020-08-22    1.0
2020-09-07    1.0
2020-08-24    1.0
2020-08-25    1.0
2020-08-26    1.0
2020-08-27    1.0
2020-08-28    1.0
2020-08-29    1.0
2020-08-30    1.0
2020-08-31    1.0
dtype: float64

```

`TextFileReader` also has a `get_chunk` method that allows you to read pieces of any size.

Write DataFrame and Series as a CSV file

Data can also be exported in a comma-separated format. With the method `pandas.DataFrame.to_csv` we can write the data into a comma-separated file:

```

[19]: df.to_csv("out.csv")

```

Of course, other delimiters can also be used, for example to write to `sys.stdout`, so that the text result is output on the console and not in a file:

```
[20]: import sys
```

```
[21]: df.to_csv(sys.stdout, sep="|")
```

```
|Title|Language|Authors|License|Publication date|doi
0|Python basics|en|Veit Schiele|BSD-3-Clause|2021-10-28|
1|Jupyter Tutorial|en|Veit Schiele|BSD-3-Clause|2019-06-27|
2|Jupyter Tutorial|de|Veit Schiele|BSD-3-Clause|2020-10-26|
3|PyViz Tutorial|en|Veit Schiele|BSD-3-Clause|2020-04-13|
```

Missing values appear in the output as empty strings. You may want to mark them with a different placeholder:

```
[22]: df.to_csv(sys.stdout, na_rep="NaN")
```

```
,Title,Language,Authors,License,Publication date,doi
0,Python basics,en,Veit Schiele,BSD-3-Clause,2021-10-28,NaN
1,Jupyter Tutorial,en,Veit Schiele,BSD-3-Clause,2019-06-27,NaN
2,Jupyter Tutorial,de,Veit Schiele,BSD-3-Clause,2020-10-26,NaN
3,PyViz Tutorial,en,Veit Schiele,BSD-3-Clause,2020-04-13,NaN
```

If no other options are given, both the row and column labels are written. Both can be deactivated:

```
[23]: df.to_csv(sys.stdout, index=False, header=False)
```

```
Python basics,en,Veit Schiele,BSD-3-Clause,2021-10-28,
Jupyter Tutorial,en,Veit Schiele,BSD-3-Clause,2019-06-27,
Jupyter Tutorial,de,Veit Schiele,BSD-3-Clause,2020-10-26,
PyViz Tutorial,en,Veit Schiele,BSD-3-Clause,2020-04-13,
```

You can also write only a subset of the columns, in an order of your choosing:

```
[24]: df.to_csv(
    sys.stdout,
    index=False,
    columns=["Title", "Language", "Authors", "Publication date"],
)
```

```
Title,Language,Authors,Publication date
Python basics,en,Veit Schiele,2021-10-28
Jupyter Tutorial,en,Veit Schiele,2019-06-27
Jupyter Tutorial,de,Veit Schiele,2020-10-26
PyViz Tutorial,en,Veit Schiele,2020-04-13
```

Working with the csv module of Python

Most forms of table data can be loaded using functions such as `pandas.read_csv`. However, in some cases manual processing may be required. It is not uncommon to receive a file with one or more incorrect rows that cause `read_csv` to fail. For any file with a single-digit delimiter, you can use Python's built-in `csv` module. To use it, pass an open file or file-like object to `csv.reader`:

```
[25]: import csv
```

```
f = open("out.csv")
```

(continues on next page)

(continued from previous page)

```

reader = csv.reader(f)

for line in reader:
    print(line)

['', 'Title', 'Language', 'Authors', 'License', 'Publication date', 'doi']
['0', 'Python basics', 'en', 'Veit Schiele', 'BSD-3-Clause', '2021-10-28', '']
['1', 'Jupyter Tutorial', 'en', 'Veit Schiele', 'BSD-3-Clause', '2019-06-27', '']
['2', 'Jupyter Tutorial', 'de', 'Veit Schiele', 'BSD-3-Clause', '2020-10-26', '']
['3', 'PyViz Tutorial', 'en', 'Veit Schiele', 'BSD-3-Clause', '2020-04-13', '']

```

Dialekte

csv-Dateien gibt es in vielen verschiedenen Varianten. Das Python csv-Modul kommt bereits mit drei verschiedenen Dialekten:

Parameter	excel	excel-tab	unix
delimiter	','	'\\t'	','
quotechar	'\"'	'\"'	'\"'
doublequote	True	True	True
skipinitialspace	False	False	False
lineterminator	'\\r\\n'	'\\r\\n'	'\\n'
quoting	csv.QUOTE_MINIMAL	csv.QUOTE_MINIMAL	csv.QUOTE_ALL
escapechar	None	None	None

You can also use it to define your own format with a different separator, a different string convention or a different end-of-line character. Registering your own dialect is recommended for this. Possible options and functions of `csv.register_dialect` are:

Argument	Description
<code>delimiter</code>	One-character string to separate fields; default value is <code>,</code> .
<code>lineterminator</code>	Line terminator for writing; default value is <code>\\r\\n</code> . Reader ignores this and recognises cross-platform line delimiters.
<code>quotechar</code>	Quotation marks for fields with special characters (like a separator); default is <code>"</code> .
<code>quoting</code>	Quoting convention. Options include <code>csv.QUOTE_ALL</code> – quote all fields, <code>csv.QUOTE_MINIMAL</code> – quote only fields with special characters like the delimiter, <code>csv.QUOTE_NONNUMERIC</code> , and <code>csv.QUOTE_NONE</code> – no quotes. The default value is <code>QUOTE_MINIMAL</code> .
<code>skipinitialspace</code>	Ignore spaces after each delimiter; default is <code>False</code> .
<code>doublequote</code>	if <code>True</code> , quotes are doubled within a field.
<code>escapechar</code>	String to bypass the delimiter when quoting is set to <code>csv.QUOTE_NONE</code> ; default is disabled.

```

[26]: csv.register_dialect(
        "my_csv_dialect",
        lineterminator="\\n",
        delimiter=";",
        quotechar="'",

```

(continues on next page)

(continued from previous page)

```
    quoting=csv.QUOTE_MINIMAL,
)
```

Now the CSV file can be opened with:

```
[27]: with open("out.csv") as f:
        reader = csv.reader(f, "my_csv_dialect")
        for line in reader:
            print(line)

['', 'Title', 'Language', 'Authors', 'License', 'Publication date', 'doi']
['0', 'Python basics', 'en', 'Veit Schiele', 'BSD-3-Clause', '2021-10-28', '']
['1', 'Jupyter Tutorial', 'en', 'Veit Schiele', 'BSD-3-Clause', '2019-06-27', '']
['2', 'Jupyter Tutorial', 'de', 'Veit Schiele', 'BSD-3-Clause', '2020-10-26', '']
['3', 'PyViz Tutorial', 'en', 'Veit Schiele', 'BSD-3-Clause', '2020-04-13', '']
```

Then we can create a Dict with data columns by using [Dict Comprehensions](#) and iterating over the values from values with `zip`. Note that this requires a lot of storage space for large files, as the rows are converted into columns:

```
[28]: with open("out.csv") as f:
        reader = csv.reader(f, "my_csv_dialect")
        lines = list(reader)
        header, values = lines[0], lines[1:]
        data_dict = {h: v for h, v in zip(header, zip(*values))}
```

```
data_dict
```

```
[28]: {'': ('0', '1', '2', '3'),
      'Title': ('Python basics',
                'Jupyter Tutorial',
                'Jupyter Tutorial',
                'PyViz Tutorial'),
      'Language': ('en', 'en', 'de', 'en'),
      'Authors': ('Veit Schiele', 'Veit Schiele', 'Veit Schiele', 'Veit Schiele'),
      'License': ('BSD-3-Clause', 'BSD-3-Clause', 'BSD-3-Clause', 'BSD-3-Clause'),
      'Publication date': ('2021-10-28', '2019-06-27', '2020-10-26', '2020-04-13'),
      'doi': ('', '', '', '')}
```

To write files with separators manually, you can use `csv.writer`. It accepts an open, writable file object and the same dialect and format options as `csv.reader`:

```
[29]: with open("new.csv", "w") as f:
        writer = csv.writer(f, "my_csv_dialect")
        writer.writerow(("", "Titel", "Sprache", "Autor*innen"))
        writer.writerow(("1", "Python basics", "en", "Veit Schiele"))
        writer.writerow(("2", "Jupyter Tutorial", "en", "Veit Schiele"))
```

```
[30]: list(open("new.csv"))
```

```
[30]: ['',Titel,Sprache,Autor*innen\n',
      '1,Python basics,en,Veit Schiele\n',
      '2,Jupyter Tutorial,en,Veit Schiele\n']
```

3.3.3 JSON

Overview

Data structure support	+/-	JSON supports array and map or object structures and many different data types including strings, numbers, boolean, null etc., but no date formats. However, JSON does not support all data types of JavaScript: NaN and Infinity become null. Note that the JSON syntax also don't support comments and you have to work around for example with a <code>__comment__</code> key/value pair.
Standardisation	+	JSON has a formal strongly typed standard (see also RFC 8259). However, JSON data also contains some pitfalls due to the ambiguity of the JSON specifications: <i>A JSON parser MUST accept all texts that conform to the JSON grammar (RFC 7159)</i> and <i>An implementation may set limits on the size of texts that it accepts. An implementation may set limits on the maximum depth of nesting. An implementation may set limits on the range and precision of numbers. An implementation may set limits on the length and character contents of strings (RFC 7158#section-9).</i> Unfortunately there is neither a reference implementation nor an official test suite that would show the expected behaviour – at least JSON_Checker gives some hints.
Schema IDL	+/-	Partly with JSON Schema Proposal , JSON Encoding Rules (JER) , Kwalify , Rx , JSON-LD or JMES-Path . After all, there are many different validators available.
Language support	++	The JSON format is very well supported by almost every programming language. The data structure of JSON closely represent objects in many languages for example a Python dict can be represented as JSON object, and a Python list by a JSON array.
Human readability	+/-	JSON is a human-readable serialisation format but it does not support comments.
Speed	++	JSON is one of the fastest human-readable formats to serialise and deserialise.
File size	+/-	JSON is in the medium range similar to YAML and TOML .

See also:

- [JC – JSON Convert](#)
- [fx](#)
- [gron](#)
- [python-json-patch](#)

Example

Response of the *OSM-Nominatim-API*

```
[
  {
    'place_id': 234847916,
    'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. https://osm.org/
↪copyright',
    'osm_type': 'relation',
    'osm_id': 131761,
    'boundingbox': ['52.5200695', '52.5232601', '13.4103097', '13.4160798'],
    'lat': '52.5216706500000004',
    'lon': '13.413278026558228',
    'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
    'class': 'highway',
    'type': 'pedestrian',
    'importance': 0.6914982526373583
  },
  {
    'place_id': 53256307,
    'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. https://osm.org/
↪copyright',
    'osm_type': 'node',
    'osm_id': 4389211800,
    'boundingbox': ['52.5231653', '52.5232653', '13.414475', '13.414575'],
    'lat': '52.5232153',
    'lon': '13.414525',
    'display_name': 'Alexanderplatz, Alexanderstra e, Mitte, Berlin, 10178,
↪Deutschland',
    'class': 'highway',
    'type': 'bus_stop',
    'importance': 0.221000000000000003,
    'icon': 'https://nominatim.openstreetmap.org/images/mapicons/transport_bus_stop2.
↪p.20.png'
  },
  {
    'place_id': 90037579,
    'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. https://osm.org/
↪copyright',
    'osm_type': 'way',
    'osm_id': 23853138,
    'boundingbox': ['52.5214702', '52.5217276', '13.4037885', '13.4045026'],
    'lat': '52.5215991',
    'lon': '13.404112295159964',
    'display_name': 'Alexander Plaza, 1, Rosenstra e, Mitte, Berlin, 10178,
↪Deutschland',
    'class': 'tourism',
    'type': 'hotel',
    'importance': 0.111000000000000002,
    'icon': 'https://nominatim.openstreetmap.org/images/mapicons/accommodation_
↪hotel2.p.20.png'
  }
]
```

JSON example

JSON (short for *JavaScript Object Notation*) has become one of the standard formats for transmitting data via HTTP request between web browsers and other applications.

JSON is similar to Python code, except for the null value and the prohibition of commas at the end of lists. The basic types are objects (dicts), arrays (lists), strings, numbers, Boolean values and null. All keys of an object must be strings. There are several Python libraries for reading and writing JSON data. I will use `json` from the Python standard library here. To convert a JSON string into Python form, I use `json.loads`:

```
[1]: import json

f = open("books.json")
data = json.load(f)

for i in data:
    print(i)

{'Title': 'Python basics', 'Language': 'en', 'Authors': 'Veit Schiele', 'License': 'BSD-
↳ 3-Clause', 'Publication date': '2021-10-28'}
{'Title': 'Jupyter Tutorial', 'Language': 'en', 'Authors': 'Veit Schiele', 'License':
↳ 'BSD-3-Clause', 'Publication date': '2019-06-27'}
{'Title': 'Jupyter Tutorial', 'Language': 'de', 'Authors': 'Veit Schiele', 'License':
↳ 'BSD-3-Clause', 'Publication date': '2020-10-26'}
{'Title': 'PyViz Tutorial', 'Language': 'en', 'Authors': 'Veit Schiele', 'License': 'BSD-
↳ 3-Clause', 'Publication date': '2020-04-13'}
```

`json.dumps`, on the other hand, converts a Python object back to JSON:

```
[2]: json.dumps(data)

[2]: '[{"Title": "Python basics", "Language": "en", "Authors": "Veit Schiele", "License":
↳ "BSD-3-Clause", "Publication date": "2021-10-28"}, {"Title": "Jupyter Tutorial",
↳ "Language": "en", "Authors": "Veit Schiele", "License": "BSD-3-Clause", "Publication_
↳ date": "2019-06-27"}, {"Title": "Jupyter Tutorial", "Language": "de", "Authors": "Veit_
↳ Schiele", "License": "BSD-3-Clause", "Publication date": "2020-10-26"}, {"Title":
↳ "PyViz Tutorial", "Language": "en", "Authors": "Veit Schiele", "License": "BSD-3-Clause
↳ ", "Publication date": "2020-04-13"}]'
```

How you convert a JSON object or list of objects into a DataFrame or other data structure for analysis is up to you. Conveniently, you can pass a list of dicts (which were previously JSON objects) to the DataFrame constructor:

```
[3]: import pandas as pd

df = pd.DataFrame(data)

df

[3]:
```

	Title	Language	Authors	License	Publication date
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

`pandas.read_json` can automatically convert JSON records in certain arrangements into a Series or DataFrame, for example:

```
[4]: df2 = pd.read_json("books.json")
```

df2

```
[4]:
```

	Title	Language	Authors	License	Publication date
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

The default options for `pandas.read_json` assume that each object in the JSON array is a row in the table.

If you want to export data from pandas to JSON, you can use `pandas.DataFrame.to_json`:

```
[5]: print(df.to_json())
```

```
{
  "Title": {
    "0": "Python basics",
    "1": "Jupyter Tutorial",
    "2": "Jupyter Tutorial",
    "3": "PyViz Tutorial"
  },
  "Language": {
    "0": "en",
    "1": "en",
    "2": "de",
    "3": "en"
  },
  "Authors": {
    "0": "Veit Schiele",
    "1": "Veit Schiele",
    "2": "Veit Schiele",
    "3": "Veit Schiele"
  },
  "License": {
    "0": "BSD-3-Clause",
    "1": "BSD-3-Clause",
    "2": "BSD-3-Clause",
    "3": "BSD-3-Clause"
  },
  "Publication date": {
    "0": "2021-10-28",
    "1": "2019-06-27",
    "2": "2020-10-26",
    "3": "2020-04-13"
  }
}
```

```
[6]: print(df.to_json(orient="records"))
```

```
[
  {
    "Title": "Python basics",
    "Language": "en",
    "Authors": "Veit Schiele",
    "License": "BSD-3-Clause",
    "Publication date": "2021-10-28"
  },
  {
    "Title": "Jupyter Tutorial",
    "Language": "en",
    "Authors": "Veit Schiele",
    "License": "BSD-3-Clause",
    "Publication date": "2019-06-27"
  },
  {
    "Title": "Jupyter Tutorial",
    "Language": "de",
    "Authors": "Veit Schiele",
    "License": "BSD-3-Clause",
    "Publication date": "2020-10-26"
  },
  {
    "Title": "PyViz Tutorial",
    "Language": "en",
    "Authors": "Veit Schiele",
    "License": "BSD-3-Clause",
    "Publication date": "2020-04-13"
  }
]
```

3.3.4 Excel

pandas also supports reading table data stored in Excel 2003 (and higher) files, either with the `ExcelFile` class or the `pandas.read_excel` function. Internally, these tools use the add-on packages `xlrd` and `openpyxl` to read XLS and XLSX files respectively. These must be installed separately from pandas with `pipenv`.

To use `ExcelFile`, create an instance by passing a path to an xls or xlsx file:

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

```
[2]: xlsx = pd.ExcelFile("library.xlsx")
```

You can then display the sheets of the file with:

```
[3]: xlsx.sheet_names
```

```
[3]: ['books']
```

```
[4]: books = pd.read_excel(xlsx, "books")
```

books

```
[4]:
```

	Titel	Sprache	Autor*innen	Lizenz	Veröffentlichungsdatum
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

If you are reading in multiple sheets of a file, it is quicker to create the Excel file, but you can also just pass the file name to `pandas.read_excel`:

```
[5]: pd.read_excel("library.xlsx", "books")
```

```
[5]:
```

	Titel	Sprache	Autor*innen	Lizenz	Veröffentlichungsdatum
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

To write pandas data in Excel format, you must first create an `ExcelWriter` and then write data to it using `pandas.DataFrame.to_excel`:

```
[6]: writer = pd.ExcelWriter("library.xlsx")
books.to_excel(writer, "books")
writer.close()
```

You can also pass a file path to `to_excel` and thus bypass the `ExcelWriter`:

```
[7]: books.to_excel("library.xlsx")
```

3.3.5 XML/HTML

Overview

Data structure support	++	XML is very flexible as each element can have attributes and arbitrary child elements.
Standardisation	++	XML is well standardised, the specification can be found at https://www.w3.org/TR/xml/ . XML supports both DOM style and streaming SAX style parsers.
Schema-IDL	++	XML schema , RELAX NG
Language support	+	Supported in all major languages, usually with built-in libraries.
Human readability	+/-	XML is a human-readable serialisation protocol. One disadvantage of XML is its verbosity, in particular its descriptive end tags.
Speed	+	XML is quite fast, although typically slower to parse than JSON.
File size	--	XML has the largest file size in comparison.

Example

Listing 1: books.xml

```
<?xml version="1.0"?>

<!--
SPDX-FileCopyrightText: 2022 Veit Schiele

SPDX-License-Identifier: BSD-3-Clause
-->

<catalog>
  <book id="1">
    <title>Python basics</title>
    <language>en</language>
    <author>Veit Schiele</author>
    <license>BSD-3-Clause</license>
    <date>2021-10-28</date>
  </book>
  <book id="2">
    <title>Jupyter Tutorial</title>
    <language>en</language>
    <author>Veit Schiele</author>
    <license>BSD-3-Clause</license>
    <date>2019-06-27</date>
  </book>
  <book id="3">
    <title>Jupyter Tutorial</title>
    <language>de</language>
    <author>Veit Schiele</author>
    <license>BSD-3-Clause</license>
    <date>2020-10-26</date>
  </book>
  <book id="4">
    <title>PyViz Tutorial</title>
    <language>en</language>
    <author>Veit Schiele</author>
    <license>BSD-3-Clause</license>
    <date>2020-04-13</date>
  </book>
</catalog>
```

See also:

- [Home](#)
- [Specification](#)
- [Validator](#)
- [The XML FAQ](#)

XML/HTML examples

HTML

Python has numerous libraries for reading and writing data in the ubiquitous HTML and XML formats. Examples are *lxml*, *Beautiful Soup* and *html5lib*. While *lxml* is generally comparatively much faster, the other libraries are better at handling corrupted HTML or XML files.

pandas has a built-in function, `read_html`, which uses libraries like *lxml*, *html5lib* and *Beautiful Soup* to automatically parse tables from HTML files as *DataFrame* objects. These have to be installed additionally. With *Spack* you can provide *lxml*, *BeautifulSoup* and *html5lib* in your kernel:

```
$ spack env activate python-311
$ spack install py-lxml py-beautifulsoup4~html5lib~lxml py-html5lib
```

Alternatively, you can install *BeautifulSoup* with other package managers, for example

```
$ pipenv install lxml beautifulsoup4 html5lib
```

To show how this works, I use an HTML file from Wikipedia that gives an overview of different serialisation formats.

```
[1]: import pandas as pd

tables = pd.read_html("https://en.wikipedia.org/wiki/Comparison_of_data_serialization_
↪formats")
```

The `pandas.read_html` function has a number of options, but by default it looks for and tries to parse all table data contained in `<table>` tags. The result is a list of *DataFrame* objects:

```
[2]: len(tables)
```

```
[2]: 3
```

```
[3]: formats = tables[0]
```

```
formats.head()
```

```
[3]:
```

	Name	Creator-maintainer \
0	Apache Avro	Apache Software Foundation
1	Apache Parquet	Apache Software Foundation
2	ASN.1	ISO, IEC, ITU-T
3	Bencode	Bram Cohen (creator) BitTorrent, Inc. (maintai...
4	Binn	Bernardo Ramos

	Based on Standardized?[definition needed] \
0	- No
1	- No
2	- Yes
3	- De facto as BEP
4	JSON (loosely) No

	Specification \
0	Apache Avro™ Specification

(continues on next page)

(continued from previous page)

```

1                               Apache Parquet
2 ISO/IEC 8824 / ITU-T X.680 (syntax) and ISO/IE...
3     Part of BitTorrent protocol specification
4                               Binn Specification

                               Binary? \
0                               Yes
1                               Yes
2     BER, DER, PER, OER, or custom via ECN
3 Except numbers and delimiters, being ASCII
4                               Yes

                               Human-readable? Supports references?e Schema-IDL? \
0                               Partialg - Built-in
1                               No No -
2 XER, JER, GSER, or custom via ECN Yesf Built-in
3                               No No No
4                               No No No

                               Standard APIs Supports zero-copy operations
0 C, C#, C++, Java, PHP, Python, Ruby -
1                               Java, Python, C++ No
2                               - OER
3                               No No
4                               No Yes

```

From here we can do some *data cleansing and analysis*, such as the number of different schema IDLs:

```
[4]: formats["Schema-IDL?"].value_counts()
```

```
[4]: Schema-IDL?
```

```

No
  ↳ 15
Yes
  ↳ 5
Built-in
  ↳ 4
Schema WD
  ↳ 1
Partial (Kwalify, Rx, built-in language type-defs)
  ↳ 1
XML schema, RELAX NG
  ↳ 1
WSDL, XML schema
  ↳ 1
Partial (JSON Schema Proposal, other JSON schemas/IDLs)
  ↳ 1
?
  ↳ 1
Ion schema
  ↳ 1
Partial (JSON Schema Proposal, ASN.1 with JER, Kwalify, Rx, Itemscrip Schema), JSON-LD
  ↳ 1

```

(continues on next page)

(continued from previous page)

```

-
→ 1
XML schema
→ 1
XML Schema
→ 1
Partial (Signature strings)
→ 1
CDDL
→ 1
Schema-IDL?
→ 1
Name: count, dtype: int64

```

XML

pandas has a function `read_xml`, which makes reading XML files very easy:

```
[5]: pd.read_xml("books.xml")
```

```
[5]:
```

	id	title	language	author	license	date
0	1	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	2	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	3	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	4	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

lxml

Alternatively, `lxml.objectify` can be used first to parse XML files. In doing so, we get a reference to the root node of the XML file with `getroot`:

```
[6]: from lxml import objectify
```

```

parsed = objectify.parse(open("books.xml"))
root = parsed.getroot()

```

```
[7]: books = []
```

```

for element in root.book:
    data = {}
    for child in element.getchildren():
        data[child.tag] = child.pyval
    books.append(data)

```

```
[8]: pd.DataFrame(books)
```

```
[8]:
```

	id	title	language	author	license	date
0	1	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	2	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27

(continues on next page)

(continued from previous page)

2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

BeautifulSoup

```
[1]: import requests
```

```
url = "https://de.wikipedia.org/wiki/Liste_der_Stra%C3%9Fen_und_Pl%C3%A4tze_in_Berlin-
↪Mitte"
r = requests.get(url)
```

1. Install:

With *Spack* you can make BeautifulSoup available in your kernel:

```
$ spack env activate python-311
$ spack install py-beautifulsoup4~html5lib~lxml
```

Alternatively, you can install BeautifulSoup with other package managers, for example

```
$ pipenv install beautifulsoup4
```

2. With `r.content` we can output the HTML of the page.

3. Next, we have to decompose this string into a Python representation of the page with BeautifulSoup:

```
[2]: from bs4 import BeautifulSoup
```

```
soup = BeautifulSoup(r.content, "html.parser")
```

4. To structure the code, we create a new function `get_dom` (**D**ocument **O**bject **M**odel) that includes all the previous code:

```
[3]: def get_dom(url):
    r = request.get(url)
    r.raise_for_status()
    return BeautifulSoup(r.content, "html.parser")
```

Filtering out individual elements can be done, for example, via CSS selectors. These can be determined in a website, for example, by right-clicking on one of the table cells in the first column of the table in Firefox. In the Inspector that now opens, you can right-click the element again and then select *Copy* → *CSS Selector*. The clipboard will then contain, for example, `table.wikitable:nth-child(13) > tbody:nth-child(2) > tr:nth-child(1)`. We now clean up this CSS selector, as we do not want to filter for the 13th child element of the `table.wikitable` or the 2nd child element in `tbody`, but only for the 1st column within `tbody`.

Finally, with `limit=3` in this notebook, we only display the first three results as an example:

```
[4]: links = soup.select(
    "table.wikitable > tbody > tr > td:nth-child(1) > a", limit=3
)
```

(continues on next page)

(continued from previous page)

```
print(links)

[<a href="/wiki/Ackerstra%C3%9Fe_(Berlin)" title="Ackerstraße (Berlin)">Ackerstraße</a>,
↪ <a href="/wiki/Alexanderplatz" title="Alexanderplatz">Alexanderplatz</a>, <a href="/
↪ wiki/Almstadtstra%C3%9Fe" title="Almstadtstraße">Almstadtstraße</a>]
```

However, we do not want the entire HTML link, but only its text content:

```
[5]: for content in links:
      print(content.text)
```

```
Ackerstraße
Alexanderplatz
Almstadtstraße
```

See also

- [Beautiful Soup Documentation](#)

3.3.6 YAML

Overview

Data structure support	++	YAML, short for <i>YAML Ain't Markup Language</i> , supports most common data types including strings, integers, floats and dates. YAML even supports references and external data.
Standardisation	+	YAML is a strongly typed formal standard, but it's hard to find schema validators.
Schema-IDL	+/-	Partly with Kwalify , Rx and built-in language type defs.
Language support	+/-	There be libraries for the most popular languages.
Human readability	+	Basic YAML is really easy to read, however YAML's complexity can confuse a reader.
Speed	--	YAML is very slow to serialise and deserialise.
File size	+/-	YAML is in the medium range similar to JSON and TOML .

See also:

- [Home](#)
- [Specification](#)
- [YAML Validator](#)
- [StrictYAML](#)
- [What YAML features does StrictYAML remove?](#)
- [noyaml.com](#)

Example

CITATION.cff:

```
# YAML 1.2
---
cff-version: 1.1.0
message: If you use this software, please cite it as below.
authors:
  - family-names: Druskat
    given-names: Stephan
    orcid: https://orcid.org/0000-0003-4925-7248
title: "My Research Software"
version: 2.0.4
doi: 10.5281/zenodo.1234
date-released: 2017-12-18
```

You can output YAML files as Python [dictionaries](#) with:

```
[1]: import yaml

with open("CITATION.cff", "r") as file:
    cite = yaml.safe_load(file)
    print(cite)

{'cff-version': '1.1.0', 'message': 'If you use this software, please cite it
↪ as below.', 'authors': [{'family-names': 'Druskat', 'given-names': 'Stephan',
↪ 'orcid': 'https://orcid.org/0000-0003-4925-7248'}], 'title': 'My Research
↪ Software', 'version': '2.0.4', 'doi': '10.5281/zenodo.1234', 'date-released':
↪ datetime.date(2017, 12, 18)}
```

3.3.7 TOML

Overview

Data structure support	+	TOML (Tom's Obvious, Minimal Language) supports most common including strings, integers, floats and dates, but not references like YAML does.
Standardisation	++	TOML is a formal strongly typed standard.
Schema-IDL	+-	Partly with JSON Schema Everywhere
Language support	++	TOML is a relatively new serialization format and doesn't have the same broad support as JSON, CSV or XML for various programming languages.
Human readability	++	One of TOML's primary goals was to be very easy to read.
Speed	+-	TOML can be processed at medium speed.
File size	-	Only XML/HTML is less compact.

You need the Python package [toml](#) to convert TOML files into Python [Dictionaries](#). You can then load TOML files, for example with:

```
import toml

config = toml.load("pyproject.toml")
```

See also:

- [Home](#)
- [GitHub](#)
- [Wiki](#)
- [What is wrong with TOML?](#)
- [An INI critique of TOML](#)

Example

pyproject.toml

```
[tool.black]
line-length = 79

[tool.isort]
atomic=true
force_grid_wrap=0
include_trailing_comma=true
lines_after_imports=2
lines_between_types=1
multi_line_output=3
not_skip="__init__.py"
use_parentheses=true

known_first_party=["MY_FIRST_MODULE", "MY_SECOND_MODULE"]
known_third_party=["mpi4py", "numpy", "requests"]
```

For Python < 3.11 you need the Python package `toml` to convert TOML files into Python dictionaries.

For Python 3.11 you can load TOML files, for example with:

```
[1]: import tomllib

with open("pyproject.toml", "rb") as f:
    data = tomllib.load(f)

data

[1]: {'tool': {'black': {'line-length': 79},
  'isort': {'atomic': True,
  'force_grid_wrap': 0,
  'include_trailing_comma': True,
  'lines_after_imports': 2,
  'lines_between_types': 1,
```

(continues on next page)

(continued from previous page)

```
'multi_line_output': 3,
'not_skip': '__init__.py',
'use_parentheses': True,
'known_first_party': ['MY_FIRST_MODULE', 'MY_SECOND_MODULE'],
'known_third_party': ['mpi4py', 'numpy', 'requests']}]}}
```

3.3.8 Pickle

Overview

Data structure support	+-	Pickle is used to store Python object structures like <code>list</code> or <code>dict</code> in a byte stream. In contrast to <code>marshal</code> , already serialised objects are tracked so that later references are not serialised again. Recursive objects are also possible.
Standardisation	++	Pickle is defined in the Python Enhancement Proposals Proposals PEP 307 , PEP 3154 and :pep:~574 .
Schema IDL	--	No
Language support	--	Python-specific
Human readability	+-	Pickle is a binary serialisation format, but it can be easily read with Python.
Speed	+-	The pickle format can usually be serialised and deserialised quickly by Python; see also Don't pickle your data .
File size	++	Compact binary format, which can, however, be compressed even further, see also Data Compression and Archiving .

See also:

[pickle – Python object serialization](#)

Documentation of the `pickle` module

[shelve – Python object persistence](#)

Indexed databases of pickle objects

[Uwe Korn: The implications of pickling ML models](#)

Alternatives to pickle for ML models

[Ned Batchelder: Pickle's nine flaws](#)

Disadvantages of pickle and alternatives

Pickle examples

Python pickle module

In this example we want to use the Python `pickle` module to save the following dict in pickle format:

```
[1]: pyviz = {
    "Title": "PyViz Tutorial",
    "Language": "de",
    "Authors": "Veit Schiele",
    "License": "BSD-3-Clause",
    "Publication date": "2020-04-13",
}
```

```
[2]: import pickle
```

```
[3]: with open("pyviz.pkl", "wb") as f:
    pickle.dump(pyviz, f, pickle.HIGHEST_PROTOCOL)
```

Now we read the pickle file again:

```
[4]: with open("pyviz.pkl", "rb") as f:
    pyviz = pickle.load(f)

print(pyviz)

{'Title': 'PyViz Tutorial', 'Language': 'de', 'Authors': 'Veit Schiele', 'License': 'BSD-
↪3-Clause', 'Publication date': '2020-04-13'}
```

This way we can easily store Python objects persistently.

Warning:

`pickle` can only be recommended as a short-term storage format. The problem is that the format is not guaranteed to remain stable over time; an object pickled today may not be unpickled with a later version of the library.

pandas

All pandas objects have a `to_pickle` method that writes data to disk in pickle format:

```
[5]: import pandas as pd
```

```
books = pd.read_pickle("books.pkl")
```

```
books
```

```
[5]:
```

	id	title	language	author	license	date
0	1	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	2	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	3	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	4	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

pandas objects all have a `to_pickle` method that writes the data to the hard disk in pickle format:

```
[6]: books.to_pickle("books.pkl")
```

3.3.9 Protocol Buffers (Protobuf)

Overview

Data structure support	+	Protobuf allows you to define data structures in *.proto files. Protobuf supports many primitive types, which can be combined into nested classes.
Standardisation	++	Protobuf is a strongly typed flexible standard.
Schema-IDL	++	Built-in IDL compiler
Language support	++	The protobuf format is well supported by many programming languages.
Human readability	--	Protobuf is not designed to be human readable.
Speed	++	Protobuf is very fast, especially in C++.
File size	++	Protobuf is the most compact format.

See also:

- [Home](#)
- [GitHub](#)
- [Language Guide \(proto3\)](#)
- [Buf](#)
 - [Home](#)
 - [Docs](#)
 - [GitHub](#)
- [gRPC](#)

3.3.10 Other Formats

Apache Avro

A compact and fast binary data format.

See also:

- [Data Serialization and Deserialization](#)

BSON

Short for *Binary JSON*. A binary data format mainly for *MongoDB*

See also:

- [Specification](#)
- [MongoDB Extended JSON](#)
- [bsondump](#)

Cap'n Proto

A fast data interchange format.

See also:

- [GitHub](#)

JSON5

A superset of JSON by including strings with multiple lines and character escapes, hexadecimal numbers, comments etc.

See also:

- [PyPI](#)

HOCON

Short for *Human-Optimized Config Object Notation*. A JSON superset with comments, multi-line strings etc.

See also:

- [GitHub](#)
- [Play framework configuration file syntax and features](#)

MessagePack

An efficient binary serialization format supported by [Redis](#) scripting.

See also:

- [Specification](#)
- [GitHub](#)

SDLang

Short for *Simple Declarative Language*. Textually represent data in a XML-like structure.

See also:

- [Language Guide](#)
- [GitHub](#)

XDR (RFC 4506)

Short for *External Data Representation Standard*. Useful for transferring data between different computer architectures.

3.4 Intake

Intake makes it easy to find, explore, load, and distribute data. Therefore it is not only interesting for data scientists and engineers, but also for data providers.

See also:

- [Docs](#)
- [GitHub](#)
- [Intake: Taking the Pain out of Data Access](#)
- [Intake: Parsing Data from Filenames and Paths](#)
- [Intake: Discovering and Exploring Data in a Graphical Interface](#)
- [Accessing Remote Data with a Generalized File System](#)

- [Intake: Caching Data on First Read Makes Future Analysis Faster](#)

3.4.1 Install Intake

Requirements

Current versions of Bokeh2.0 and Panel must be available in order to use *intake.gui*.

Installation

Intake can be easily installed for your Jupyter kernel with:

```
$ pipenv install intake
```

Create a catalog with sample data

For the following examples we need some data sets that we create with:

```
$ pipenv run intake example
Creating example catalog...
  Writing us_states.yml
  Writing states_1.csv
  Writing states_2.csv

To load the catalog:
  >>> import intake
  >>> cat = intake.open_catalog('us_states.yml')
```

3.4.2 Intake for data scientists

Intake makes it easy to load many different formats and types. For a complete overview, take a look at the [Plugin Directory](#) and the [Intake Project Dashboard](#). Intake then transfers the data to common storage formats such as Pandas DataFrames, Numpy arrays or Python lists. They are then easily searchable and also accessible to distributed systems. If you are missing a plugin, you can also order one yourself, as described in [Making Drivers](#).

Load a data source

Hereinafter we will read two csv data records and transfer them to an intake catalog.

```
[1]: import intake

ds = intake.open_csv("states_*.csv")

print(ds)

sources:
  csv:
    args:
```

(continues on next page)

(continued from previous page)

```

urlpath: states_*.csv
description: ''
driver: intake.source.csv.CSVSource
metadata: {}

```

Mit der `open_*`-Funktion von Intake lassen sich verschiedenen Datenquellen einlesen. Je nach Datenformat oder Dienst lassen sich unterschiedliche Argmumente verwenden.

Configure the search path for data sources

Intake checks the Intake configuration file for `catalog_path` and the environment variable "INTAKE_PATH" for a colon-separated list of paths or semicolons in Windows to look for catalog files. When importing `intake`, all entries from all catalogs that are referenced by `intake.cat` as part of a global catalog are displayed.

Read data

Intake reads data in containers of various formats:

- Tables in Pandas DataFrames
- Multi-dimensional arrays in numpy arrays
- Semi-structured data in Python lists of objects, usually dictionaries

To find out in which container format Intake holds the data, you can use the `container` attribute:

```
[2]: ds.container
```

```
[2]: 'dataframe'
```

In addition to `dataframe`, the result can also be `ndarray` or `python`.

```
[3]: df = ds.read()
```

```
df.head()
```

```
[3]:
```

	state	slug	code	nickname	\
0	Alabama	alabama	AL	Yellowhammer State	
1	Alaska	alaska	AK	The Last Frontier	
2	Arizona	arizona	AZ	The Grand Canyon State	
3	Arkansas	arkansas	AR	The Natural State	
4	California	california	CA	Golden State	

	website	admission_date	admission_number	capital_city	\
0	http://www.alabama.gov	1819-12-14	22	Montgomery	
1	http://alaska.gov	1959-01-03	49	Juneau	
2	https://az.gov	1912-02-14	48	Phoenix	
3	http://arkansas.gov	1836-06-15	25	Little Rock	
4	http://www.ca.gov	1850-09-09	31	Sacramento	

	capital_url	population	population_rank	\
0	http://www.montgomeryal.gov	4833722	23	
1	http://www.juneau.org	735132	47	

(continues on next page)

(continued from previous page)

```

2      https://www.phoenix.gov      6626624      15
3      http://www.littlerock.org      2959373      32
4      http://www.cityofsacramento.org      38332521      1

```

```

                                constitution_url \
0      http://alisondb.legislature.state.al.us/alison...
1      http://www.legis.state.ak.us/basis/folioproxy...
2      http://www.azleg.gov/Constitution.asp
3      http://www.arkleg.state.ar.us/assembly/Summary...
4      http://www.leginfo.ca.gov/const-toc.html

```

```

                                state_flag_url \
0      https://cdn.civil.services/us-states/flags/ala...
1      https://cdn.civil.services/us-states/flags/ala...
2      https://cdn.civil.services/us-states/flags/ari...
3      https://cdn.civil.services/us-states/flags/ark...
4      https://cdn.civil.services/us-states/flags/cal...

```

```

                                state_seal_url \
0      https://cdn.civil.services/us-states/seals/ala...
1      https://cdn.civil.services/us-states/seals/ala...
2      https://cdn.civil.services/us-states/seals/ari...
3      https://cdn.civil.services/us-states/seals/ark...
4      https://cdn.civil.services/us-states/seals/cal...

```

```

                                map_image_url \
0      https://cdn.civil.services/us-states/maps/alab...
1      https://cdn.civil.services/us-states/maps/alas...
2      https://cdn.civil.services/us-states/maps/ariz...
3      https://cdn.civil.services/us-states/maps/arka...
4      https://cdn.civil.services/us-states/maps/cali...

```

```

                                landscape_background_url \
0      https://cdn.civil.services/us-states/backgroun...
1      https://cdn.civil.services/us-states/backgroun...
2      https://cdn.civil.services/us-states/backgroun...
3      https://cdn.civil.services/us-states/backgroun...
4      https://cdn.civil.services/us-states/backgroun...

```

```

                                skyline_background_url \
0      https://cdn.civil.services/us-states/backgroun...
1      https://cdn.civil.services/us-states/backgroun...
2      https://cdn.civil.services/us-states/backgroun...
3      https://cdn.civil.services/us-states/backgroun...
4      https://cdn.civil.services/us-states/backgroun...

```

```

                                twitter_url \
0      https://twitter.com/alabamagov
1      https://twitter.com/alaska
2      NaN
3      https://twitter.com/arkansasgov
4      https://twitter.com/cagovernment

```

(continues on next page)

(continued from previous page)

```

                                facebook_url
0          https://www.facebook.com/alabamagov
1 https://www.facebook.com/AlaskaLocalGovernments
2                                NaN
3          https://www.facebook.com/Arkansas.gov
4                                NaN

```

```
[4]: for chunk in ds.read_chunked():
      print("Chunk: %d" % len(chunk))
```

```

Chunk: 24
Chunk: 26

```

```
[5]: ddf = ds.to_dask()
```

```
ddf.head()
```

```
[5]:
      state      slug code      nickname \
0   Alabama  alabama  AL  Yellowhammer State
1   Alaska   alaska  AK    The Last Frontier
2   Arizona  arizona  AZ  The Grand Canyon State
3   Arkansas arkansas  AR    The Natural State
4  California california  CA      Golden State

      website admission_date admission_number capital_city \
0 http://www.alabama.gov  1819-12-14          22  Montgomery
1   http://alaska.gov    1959-01-03          49    Juneau
2   https://az.gov       1912-02-14          48    Phoenix
3   http://arkansas.gov  1836-06-15          25  Little Rock
4   http://www.ca.gov    1850-09-09          31  Sacramento

      capital_url  population  population_rank \
0 http://www.montgomeryal.gov  4833722          23
1   http://www.juneau.org      735132          47
2   https://www.phoenix.gov    6626624          15
3   http://www.littlerock.org   2959373          32
4 http://www.cityofsacramento.org 38332521          1

      constitution_url \
0 http://alisondb.legislature.state.al.us/alison...
1 http://www.legis.state.ak.us/basis/folioproxy...
2   http://www.azleg.gov/Constitution.asp
3 http://www.arkleg.state.ar.us/assembly/Summary...
4   http://www.leginfo.ca.gov/const-toc.html

      state_flag_url \
0 https://cdn.civil.services/us-states/flags/ala...
1 https://cdn.civil.services/us-states/flags/ala...
2 https://cdn.civil.services/us-states/flags/ari...
3 https://cdn.civil.services/us-states/flags/ark...
4 https://cdn.civil.services/us-states/flags/cal...
```

(continues on next page)

(continued from previous page)

```

                                state_seal_url \
0 https://cdn.civil.services/us-states/seals/ala...
1 https://cdn.civil.services/us-states/seals/ala...
2 https://cdn.civil.services/us-states/seals/ari...
3 https://cdn.civil.services/us-states/seals/ark...
4 https://cdn.civil.services/us-states/seals/cal...

                                map_image_url \
0 https://cdn.civil.services/us-states/maps/alab...
1 https://cdn.civil.services/us-states/maps/alas...
2 https://cdn.civil.services/us-states/maps/ariz...
3 https://cdn.civil.services/us-states/maps/arka...
4 https://cdn.civil.services/us-states/maps/cali...

                                landscape_background_url \
0 https://cdn.civil.services/us-states/backgroun...
1 https://cdn.civil.services/us-states/backgroun...
2 https://cdn.civil.services/us-states/backgroun...
3 https://cdn.civil.services/us-states/backgroun...
4 https://cdn.civil.services/us-states/backgroun...

                                skyline_background_url \
0 https://cdn.civil.services/us-states/backgroun...
1 https://cdn.civil.services/us-states/backgroun...
2 https://cdn.civil.services/us-states/backgroun...
3 https://cdn.civil.services/us-states/backgroun...
4 https://cdn.civil.services/us-states/backgroun...

                                twitter_url \
0 https://twitter.com/alabamagov
1 https://twitter.com/alaska
2 <NA>
3 https://twitter.com/arkansasgov
4 https://twitter.com/cagovernment

                                facebook_url
0 https://www.facebook.com/alabamagov
1 https://www.facebook.com/AlaskaLocalGovernments
2 <NA>
3 https://www.facebook.com/Arkansas.gov
4 <NA>

```

```
[6]: cat = intake.open_catalog("us_states.yml")
```

```
[7]: list(cat)
```

```
[7]: ['states']
```

```
[8]: cat.states.to_dask()[["state", "slug"]].head()
```

```
[8]:      state      slug
```

(continues on next page)

(continued from previous page)

```

0 Alabama alabama
1 Alaska alaska
2 Arizona arizona
3 Arkansas arkansas
4 California california

```

```
[9]: cat.states(csv_kwargs={"header": None, "skiprows": 1}).read().head()
```

```

[9]:
      0      1      2      3      4 \
0 Alabama alabama AL Yellowhammer State http://www.alabama.gov
1 Alaska alaska AK The Last Frontier http://alaska.gov
2 Arizona arizona AZ The Grand Canyon State https://az.gov
3 Arkansas arkansas AR The Natural State http://arkansas.gov
4 California california CA Golden State http://www.ca.gov

      5      6      7      8      9      10 \
0 1819-12-14 22 Montgomery http://www.montgomeryal.gov 4833722 23
1 1959-01-03 49 Juneau http://www.juneau.org 735132 47
2 1912-02-14 48 Phoenix https://www.phoenix.gov 6626624 15
3 1836-06-15 25 Little Rock http://www.littlerock.org 2959373 32
4 1850-09-09 31 Sacramento http://www.cityofsacramento.org 38332521 1

      11 \
0 http://alisondb.legislature.state.al.us/alison...
1 http://www.legis.state.ak.us/basis/folioproxy...
2 http://www.azleg.gov/Constitution.asp
3 http://www.arkleg.state.ar.us/assembly/Summary...
4 http://www.leginfo.ca.gov/const-toc.html

      12 \
0 https://cdn.civil.services/us-states/flags/ala...
1 https://cdn.civil.services/us-states/flags/ala...
2 https://cdn.civil.services/us-states/flags/ari...
3 https://cdn.civil.services/us-states/flags/ark...
4 https://cdn.civil.services/us-states/flags/cal...

      13 \
0 https://cdn.civil.services/us-states/seals/ala...
1 https://cdn.civil.services/us-states/seals/ala...
2 https://cdn.civil.services/us-states/seals/ari...
3 https://cdn.civil.services/us-states/seals/ark...
4 https://cdn.civil.services/us-states/seals/cal...

      14 \
0 https://cdn.civil.services/us-states/maps/alab...
1 https://cdn.civil.services/us-states/maps/alas...
2 https://cdn.civil.services/us-states/maps/ariz...
3 https://cdn.civil.services/us-states/maps/arka...
4 https://cdn.civil.services/us-states/maps/cali...

      15 \
0 https://cdn.civil.services/us-states/backgroun...

```

(continues on next page)

(continued from previous page)

```

1 https://cdn.civil.services/us-states/backgroun...
2 https://cdn.civil.services/us-states/backgroun...
3 https://cdn.civil.services/us-states/backgroun...
4 https://cdn.civil.services/us-states/backgroun...

                                16 \
0 https://cdn.civil.services/us-states/backgroun...
1 https://cdn.civil.services/us-states/backgroun...
2 https://cdn.civil.services/us-states/backgroun...
3 https://cdn.civil.services/us-states/backgroun...
4 https://cdn.civil.services/us-states/backgroun...

                                17 \
0 https://twitter.com/alabamagov
1 https://twitter.com/alaska
2 NaN
3 https://twitter.com/arkansasgov
4 https://twitter.com/cagovernment

                                18
0 https://www.facebook.com/alabamagov
1 https://www.facebook.com/AlaskaLocalGovernments
2 NaN
3 https://www.facebook.com/Arkansas.gov
4 NaN

```

3.4.3 Intake-GUI: Exploring data in a graphical user interface

Intake GUI has been re-implemented so that it can be made available not only in Jupyter notebooks, but also in other web applications. It displays the contents of all installed catalogs and enables local and remote catalogs to be selected and to be searched and selected from.

Intake supports the division of labor between data engineers who curate, manage, and deploy data, and data scientists who analyse and visualise data without having to know how it's stored.

The Intake GUI is based on [Panel](#), with the control panel offering a composite dashboard solution for displaying plots, images, tables, texts and widgets. Panel works both in a Jupyter notebook and in a standalone Tornado application.

From a data engineer's point of view, this means that you can deploy the recording GUI at an endpoint and use it as a data exploration tool for your data users. This also means that it's easy to adapt and reorganise the GUI in order to insert your own logo, reuse parts of it in your own applications or add new functions.

In the future, Intake-GUI should also allow the input of user parameters as well as the editing and saving of catalogs.

```
[1]: import intake
```

```
intake.gui
```

```
Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

```
[1]: Column(width_policy='max')
      [0] Row(width_policy='max')
          [0] PNG(str)
          [1] Column(width_policy='max')
              [0] Column(name='Select Catalog')
                  [0] Markdown(str, max_height=40)
                  [1] MultiSelect(min_width=200, options={'builtin': <Intake c...}, size=9,
→ value=[<Intake catalog: b...], width_policy='min')
              [1] Row(name='Controls')
                  [0] Toggle(name='', width=50)
                  [1] Button(name='-', width=50)
                  [2] Toggle(name='', width=50)
              [2] Column(width_policy='max')
                  [0] Column(name='Select Data Source')
                      [0] Markdown(str, max_height=40)
                      [1] MultiSelect(min_width=200, size=9, width_policy='min')
                  [1] Row(name='Controls')
                      [0] Toggle(disabled=True, name='', width=50)
                      [1] Toggle(disabled=True, name='', width=50)
                  [3] Column(height=240, name='Description', scroll=True, sizing_mode='stretch_
→ width', width_policy='max')
                      [0] Markdown(str)
              [1] Row(width_policy='max')
                  [0] Row(height_policy='min', max_width=5000, name='Search', width_policy='max')
                  [1] Column(max_width=5000, name='Add Catalog', width_policy='max')
                  [2] Column(name='Plot', width_policy='max')
```

The GUI contains three main areas:

1. a list of catalogs. The *builtin* catalog shown by default contains data records installed in the system, just like `intake.cat`.
2. a list of the sources in the currently selected catalog.
3. a description of the currently selected source.

Ad 1: Catalogs

No catalog is currently displayed in the list of catalogs. However, under the three main areas there are three buttons that can be used to add, remove, or search catalogs.

The buttons are also available through the API, e.g. for Add Catalog with:

```
[2]: intake.gui.add("./us_crime/us_crime.yaml")
```

Remote catalogs are e.g. available under

- https://s3.amazonaws.com/earth-data/UCMerced_LandUse/catalog.yml
- https://raw.githubusercontent.com/ContinuumIO/anaconda-package-data/master/catalog/anaconda_package_data.yaml
- <https://raw.githubusercontent.com/pangeo-data/pangeo-datastore/master/intake-catalogs/master.yaml>

Ad 2. Sources

Selecting a source from the list updates the descriptive text on the left side of the user interface.

This is also available via the API:

```
[3]: intake.gui.sources
```

```
[3]: [name: us_crime
      container: dataframe
      plugin: ['csv']
      driver: ['csv']
      description: US Crime data [UCRDataTool] (https://www.ucrdatatool.gov/Search/Crime/State/
      ↪StatebyState.cfm)
      direct_access: forbid
      user_parameters: []
      metadata:
        plots:
          line_example:
            kind: line
            y: ['Robbery', 'Burglary']
            x: Year
          violin_example:
            kind: violin
            y: ['Burglary rate', 'Larceny-theft rate', 'Robbery rate', 'Violent Crime rate']
            group_label: Type of crime
            value_label: Rate per 100k
            invert: True
      args:
        urlpath: {{ CATALOG_DIR }}/data/crime.csv]
```

This consists of a list of regular Intake data source entries. To look at the first entries, we can enter the following:

```
[4]: source = intake.gui.sources[0]
```

```
source.to_dask().head()
```

```

[4]: textasciitildeYear Population Violent crime total \
0      1960      179323175      288460
1      1961      182992000      289390
2      1962      185771000      301510
3      1963      188483000      316970
4      1964      191141000      364220

Murder and nonnegligent Manslaughter Legacy rape /1 Revised rape /2 \
0      9110      17190      NaN
1      8740      17220      NaN
2      8530      17550      NaN
3      8640      17650      NaN
4      9360      21420      NaN

Robbery Aggravated assault Property crime total Burglary ... \
0      107840      154320      3095700      912100 ...
1      106670      156760      3198600      949600 ...
2      110860      164570      3450700      994300 ...
3      116470      174210      3792500      1086400 ...
4      130390      203050      4200400      1213200 ...

Violent Crime rate Murder and nonnegligent manslaughter rate \
0      160.9      5.1
1      158.1      4.8
2      162.3      4.6
3      168.2      4.6
4      190.6      4.9

Legacy rape rate /1 Revised rape rate /2 Robbery rate \
0      9.6      NaN      60.1
1      9.4      NaN      58.3
2      9.4      NaN      59.7
3      9.4      NaN      61.8
4      11.2      NaN      68.2

Aggravated assault rate Property crime rate Burglary rate \
0      86.1      1726.3      508.6
1      85.7      1747.9      518.9
2      88.6      1857.5      535.2
3      92.4      2012.1      576.4
4      106.2      2197.5      634.7

Larceny-theft rate Motor vehicle theft rate
0      1034.7      183.0
1      1045.4      183.6
2      1124.8      197.4
3      1219.1      216.6
4      1315.5      247.4

[5 rows x 22 columns]

```

```

[5]: source.gui

```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

```
[5]: Column
      [0] Row
          [0] PNG(str)
          [1] Column
              [0] Toggle(name='', width=50)
              [1] Toggle(disabled=True, name='', width=50)
          [1] Column(height=240, name='Description', scroll=True, sizing_mode='stretch_width',
↳width_policy='max')
              [0] Markdown(str)
          [2] Column(name='Plot', width_policy='max')
```

```
[6]: intake.gui.source.description
```

```
[6]: Column(height=240, name='Description', scroll=True, sizing_mode='stretch_width', width_
↳policy='max')
      [0] Markdown(str)
```

```
[7]: cat = intake.open_catalog("./us_crime/us_crime.yaml")
```

```
cat.gui
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

```
[7]: Column(width_policy='max')
      [0] Row(width_policy='max')
          [0] PNG(str)
          [1] Column(width_policy='max')
              [0] Column(name='Select Catalog')
                  [0] Markdown(str, max_height=40)
                  [1] MultiSelect(min_width=200, options={'us_crime': <...>, size=9,
↳value=[<Intake catalog: u...>, width_policy='min'])
```

(continues on next page)

(continued from previous page)

```

[1] Row(name='Controls')
    [0] Toggle(name='', width=50)
    [1] Button(name='-', width=50)
    [2] Toggle(name='', width=50)
[2] Column(width_policy='max')
    [0] Column(name='Select Data Source')
        [0] Markdown(str, max_height=40)
        [1] MultiSelect(min_width=200, options=OrderedDict([('us_crime', ...)],
↪size=9, value=[name: us_crime
container:...], width_policy='min')
    [1] Row(name='Controls')
        [0] Toggle(name='', width=50)
        [1] Toggle(disabled=True, name='', width=50)
    [3] Column(height=240, name='Description', scroll=True, sizing_mode='stretch_
↪width', width_policy='max')
        [0] Markdown(str)
[1] Row(width_policy='max')
    [0] Row(height_policy='min', max_width=5000, name='Search', width_policy='max')
    [1] Column(max_width=5000, name='Add Catalog', width_policy='max')
    [2] Column(name='Plot', width_policy='max')

```

```
[8]: us_crime = cat.gui.sources[0]
```

```

[9]: intake.output_notebook()

us_crime.plot.bivariate(
    "Burglary rate",
    "Property crime rate",
    legend=False,
    width=500,
    height=400
) * us_crime.plot.scatter(
    "Burglary rate",
    "Property crime rate",
    color="black",
    size=15,
    legend=False,
) + us_crime.plot.table(
    ["Burglary rate", "Property crime rate"],
    width=350,
    height=350
)

```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

```
[9]: :Layout
      .Overlay.I :Overlay
      .Bivariate.I :Bivariate [Burglary rate,Property crime rate] (Density)
      .Scatter.I :Scatter [Burglary rate] (Property crime rate)
      .Table.I :Table [Burglary rate,Property crime rate]
```

Ad 3. Source view

As soon as catalogs are loaded and the desired sources have been selected, they are available under the attribute `intake.gui.sources`. Each source entry has methods and can be opened as a data source like any catalog entry. For *Source: UCMerced_LandUse_by_landuse*, the entry looks like this:

```
name: UCMerced_LandUse_by_landuse
container: None
plugin: []
description: All images matching given landuse from UCMerced_LandUse/Image.
direct_access: forbid
user_parameters: [{'name': 'landuse', 'description': 'which landuse to collect', 'type':
  ↳ 'str', 'default': 'airplane'}]
metadata:
args:
  urlpath: s3://earth-data/UCMerced_LandUse/Images/{ landuse }/{ landuse }{id:2d}.tif
  storage_options:
    anon: True
    concat_dim: id
    coerce_shape: [256, 256]
```

Below the list of sources there is a series of buttons for opening up the selected data source: **Plot** opens a sub-window to display the predefined (i.e. the ones specified in yaml) plots for the selected source.

See also:

- [GUI](#)

3.4.4 Intake for data engineers

Intake supports data engineers with the provision of data and with the specification of the data sources, the distribution of the data, the parameterisation of the user options etc. This makes it easier for data scientists to access the data afterwards, as the possible options are already specified in the catalog.

```
[1]: import hvplot.pandas
import intake
```

```
intake.output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Intake data sets are loaded with so-called drivers, some come with the intake package, but others have to be reloaded as [plug-ins](#). You can display the available drivers as follows:

```
[2]: list(intake.registry)
```

```
[2]: ['parquet',  
      'alias',  
      'catalog',  
      'csv',  
      'intake_remote',  
      'json',  
      'jsonl',  
      'ndzarr',  
      'numpy',  
      'textfiles',  
      'tiled',  
      'tiled_cat',  
      'yaml_file_cat',  
      'yaml_files_cat',  
      'zarr_cat']
```

Each of these drivers is assigned an `intake.open_*` function. It is also possible to refer to drivers by the fully qualified name (e.g. `package.submodule.DriverClass`). In the following example, however, we will focus on the `csv` driver that is included in the standard Intake installation.

In general, the first step in writing a catalog entry is to use the appropriate `open_*` function to create a `DataSource` object:

```
[3]: source = intake.open_csv(  
      "https://timeseries.weebly.com/uploads/" "2/1/0/8/21086414/sea_ice.csv"  
    )
```

The above specification has now created a `DataSource` object, but has not yet checked whether the data can actually be accessed. To test whether the loading was really successful, the source itself can be opened (`source.discover`) or read (`source.read`):

```
[4]: source.discover()
```

```
[4]: {'dtype': {'Time': 'object', 'Arctic': 'float64', 'Antarctica': 'float64'},  
      'shape': (None, 3),  
      'npartitions': 1,  
      'metadata': {}}
```

```
[5]: df = source.read()
```

```
df.head()
```

```
[5]:
```

	Time	Arctic	Antarctica
0	1990M01	12.72	3.27
1	1990M02	13.33	2.15
2	1990M03	13.44	2.71
3	1990M04	12.16	5.10
4	1990M05	10.84	7.37

After we have determined that the data can be loaded as desired, we want to open up the data visually:

```
[6]: df.hvplot(
      kind="line", x="Time", y=["Arctic", "Antarctica"], width=700, height=500
    )
```

```
[6]: :NdOverlay    [Variable]
      :Curve      [Time]    (value)
```

Now we can load a source correctly and also receive a graphic output for opening up the data. We can now display this recipe in the YAML syntax with:

```
[7]: print(source.yaml())
```

```
sources:
  csv:
    args:
      urlpath: https://timeseries.weebly.com/uploads/2/1/0/8/21086414/sea_ice.csv
      description: ''
    driver: intake.source.csv.CSVSource
    metadata: {}
```

Finally, we can create a YAML file containing this recipe with an additional description and the tested diagram:

```
[8]: %%writefile sea.yaml
sources:
  sea_ice:
    args:
      urlpath: "https://timeseries.weebly.com/uploads/2/1/0/8/21086414/sea_ice.csv"
      description: "Polar sea ice cover"
    driver: csv
    metadata:
      plots:
        basic:
          kind: line
          x: Time
          y: [Arctic, Antarctica]
          width: 700
          height: 500

Overwriting sea.yaml
```

To check that the YAML file works too, we can reload it and try to work with it:

```
[9]: cat = intake.open_catalog("sea.yaml")
```

```
[10]: cat.sea_ice.plot.basic()
```

```
[10]: :NdOverlay    [Variable]
      :Curve      [Time]    (value)
```

The catalog appears to be functional and can now be released. The easiest way to share an Intake catalog is to put it in a place where it can be read by your target audience. In this tutorial stored in a Git repo, this can be the url of the file in the repo. All you have to share with your users is the URL of the catalog. You can try this yourself with:

```
[11]: cat = intake.open_catalog(
      "https://raw.githubusercontent.com/veit/Python4DataScience/main/docs/data-processing/
↪intake/sea.yaml"
      )
```

```
[12]: cat.sea_ice.read().head()
```

```
[12]:      Time  Arctic  Antarctica
0  1990M01   12.72      3.27
1  1990M02   13.33      2.15
2  1990M03   13.44      2.71
3  1990M04   12.16      5.10
4  1990M05   10.84      7.37
```

Note

This catalog is also a `DataSource` instance, i.e. you can refer to it from other catalogs and thus build a hierarchy of data sources. For example, you have a master or main catalog that references several other catalogs, each with entries of a certain type and the whole thing can e.g. be searched with *Intake-GUI*. In this way, the overall data acquisition structure has a structure that makes it easier to navigate to the correct data set. You can even have separate hierarchies that reference the same data.

```
[13]: print(cat.yaml())
```

```
sources:
  sea:
    args:
      path: https://raw.githubusercontent.com/veit/Python4DataScience/main/docs/data-
↪processing/intake/sea.yaml
      description: ''
      driver: intake.catalog.local.YAMLFileCatalog
      metadata: {}
```

3.5 httpx

`httpx` is an http client with which requests can be sent easily.

3.5.1 httpx installation and sample application

Installation

The httpx library is useful for communicating with REST APIs. With *Spack* you can provide httpx in your kernel:

```
$ spack env activate python-311
$ spack install py-httpx
```

Alternatively, you can install httpx with other package managers, for example

```
$ pipenv install httpx
```

Example OSM Nominatim API

In this example we get our data from the [OpenStreetMap Nominatim API](https://nominatim.openstreetmap.org/search?). This can be reached via the URL <https://nominatim.openstreetmap.org/search?>. To e.g. receive information about the Berlin Congress Center in Berlin in JSON format, the URL <https://nominatim.openstreetmap.org/search.php?q=Alexanderplatz+Berlin&format=json> should be given, and if you want to display the corresponding map section you just have to leave out `&format=json`.

Then we define the search URL and the parameters. Nominatim expects at least the following two parameters

Key	Value
q	Address query that allows the following specifications: street, city, county, state, country and postalcode.
format	Format in which the data is returned. Possible values are html, xml, json, jsonv2, geojson and geocodejson.

The query can then be made with:

```
[1]: import httpx

search_url = "https://nominatim.openstreetmap.org/search?"
params = {
    "q": "Alexanderplatz, Berlin",
    "format": "json",
}
r = httpx.get(search_url, params=params)
```

```
[2]: r.status_code
```

```
[2]: 200
```

```
[3]: r.json()
```

```
[3]: [{'place_id': 128497332,
      'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
      'osm_type': 'way',
      'osm_id': 783052052,
      'lat': '52.5219814',
```

(continues on next page)

(continued from previous page)

```

'lon': '13.413635717448294',
'class': 'place',
'type': 'square',
'place_rank': 25,
'importance': 0.47149825263735834,
'addresstype': 'square',
'name': 'Alexanderplatz',
'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
'boundingbox': ['52.5201457', '52.5238113', '13.4103097', '13.4160801']],
{'place_id': 128243381,
 'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
 'osm_type': 'node',
 'osm_id': 3908141014,
 'lat': '52.5215661',
 'lon': '13.4112804',
 'class': 'railway',
 'type': 'station',
 'place_rank': 30,
 'importance': 0.43609907778808027,
 'addresstype': 'railway',
 'name': 'Alexanderplatz',
 'display_name': 'Alexanderplatz, Dircksenstraße, Mitte, Berlin, 10179, Deutschland',
 'boundingbox': ['52.5165661', '52.5265661', '13.4062804', '13.4162804']],
{'place_id': 128416772,
 'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
 'osm_type': 'way',
 'osm_id': 346206374,
 'lat': '52.5216214',
 'lon': '13.4131913',
 'class': 'highway',
 'type': 'pedestrian',
 'place_rank': 26,
 'importance': 0.10000999999999993,
 'addresstype': 'road',
 'name': 'Alexanderplatz',
 'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
 'boundingbox': ['52.5216214', '52.5216661', '13.4131913', '13.4131914']],
{'place_id': 127680907,
 'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
 'osm_type': 'way',
 'osm_id': 301733776,
 'lat': '52.5222454',
 'lon': '13.4158136',
 'class': 'highway',
 'type': 'primary',
 'place_rank': 26,
 'importance': 0.10000999999999993,
 'addresstype': 'road',
 'name': 'Alexanderstraße',
 'display_name': 'Alexanderstraße, Mitte, Berlin, 10178, Deutschland',
 'boundingbox': ['52.5222454', '52.5224356', '13.4153983', '13.4158136']}]

```

Three different locations are found, the square, a bus stop and a hotel. In order to be able to filter further, we can only

display the most important location:

```
[4]: params = {"q": "Alexanderplatz, Berlin", "format": "json", "limit": "1"}
r = httpx.get(search_url, params=params)
r.json()

[4]: [{'place_id': 128497332,
      'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
      'osm_type': 'way',
      'osm_id': 783052052,
      'lat': '52.5219814',
      'lon': '13.413635717448294',
      'class': 'place',
      'type': 'square',
      'place_rank': 25,
      'importance': 0.47149825263735834,
      'addresstype': 'square',
      'name': 'Alexanderplatz',
      'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
      'boundingbox': ['52.5201457', '52.5238113', '13.4103097', '13.4160801']]}
```

Clean Code

Now that we know the code works, let's turn everything into a clean and flexible function.

To ensure that the interaction was successful, we use the `raise_for_status` method of `httpx`, which throws an exception if the HTTP status code isn't 200 OK:

```
[5]: r.raise_for_status()

[5]: <Response [200 OK]>
```

Since we don't want to exceed the load limits of the Nominatim API, we will delay our `httpx` with the `time.sleep` function:

```
[6]: from time import sleep

sleep(1)
r.json()

[6]: [{'place_id': 128497332,
      'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
      'osm_type': 'way',
      'osm_id': 783052052,
      'lat': '52.5219814',
      'lon': '13.413635717448294',
      'class': 'place',
      'type': 'square',
      'place_rank': 25,
      'importance': 0.47149825263735834,
      'addresstype': 'square',
      'name': 'Alexanderplatz',
      'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
      'boundingbox': ['52.5201457', '52.5238113', '13.4103097', '13.4160801']]}
```

Next we declare the function itself. As arguments we need the address, the format, the limit of the objects to be returned with the default value 1 and further kwargs (**keyword arguments**) that are passed as parameters:

```
[7]: def nominatim_search(address, format="json", limit=1, **kwargs):
    """Thin wrapper around the Nominatim search API.
    For the list of parameters see
    https://nominatim.org/release-docs/develop/api/Search/#parameters
    """
    search_url = "https://nominatim.openstreetmap.org/search?"
    params = {"q": address, "format": format, "limit": limit, **kwargs}
    r = httpx.get(search_url, params=params)
    # Raise an exception if the status is unsuccessful
    r.raise_for_status()

    sleep(1)
    return r.json()
```

Now we can try out the function, for example with

```
[8]: nominatim_search("Alexanderplatz, Berlin")
[8]: [{'place_id': 128497332,
      'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
      'osm_type': 'way',
      'osm_id': 783052052,
      'lat': '52.5219814',
      'lon': '13.413635717448294',
      'class': 'place',
      'type': 'square',
      'place_rank': 25,
      'importance': 0.47149825263735834,
      'addresstype': 'square',
      'name': 'Alexanderplatz',
      'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
      'boundingbox': ['52.5201457', '52.5238113', '13.4103097', '13.4160801']}]
```

Caching

If the same queries are to be asked over and over again within a session, it makes sense to call up this data only once and use it again. In Python we can use `lru_cache` from Python's standard `functools` library. `lru_cache` saves the last `N` requests (**L**east **R**ecent **U**sed) and as soon as the limit is exceeded, the oldest values are discarded. To use this for the `nominatim_search` method, all you have to do is define an import and a decorator:

```
[9]: from functools import lru_cache

@lru_cache(maxsize=1000)
def nominatim_search(address, format="json", limit=1, **kwargs):
    """ . . . """
```

However, `lru_cache` only saves the results during a session. If a script terminates because of a timeout or an exception, the results are lost. If the data is to be saved more permanently, tools such as `joblib` or `python-diskcache` can be used.

3.5.2 Create module

It is not very practical to start Jupyter every time and go through all the cells of the [httpx notebook](#) just to be able to use the functions. Instead, we should store our functions in a separate module, like in [nominatim.py](#):

1. For this I have created a new text file in Jupyter in the same place as these notebooks, and named it `nominatim.py`.
2. Then I copied the imports, the method `nominatim_search` and its decorator `lru_cache` and saved the file.
3. Now we can go back to our notebook and import the code from this file and do our searches:

```
[1]: from nominatim import nominatim_search
```

```
[2]: nominatim_search("Alexanderplatz, Berlin, Germany")
```

```
[2]: [{ 'place_id': 128497332,
      'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
      'osm_type': 'way',
      'osm_id': 783052052,
      'lat': '52.5219814',
      'lon': '13.413635717448294',
      'class': 'place',
      'type': 'square',
      'place_rank': 25,
      'importance': 0.47149825263735834,
      'addresstype': 'square',
      'name': 'Alexanderplatz',
      'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
      'boundingbox': ['52.5201457', '52.5238113', '13.4103097', '13.4160801']}]
```

The outsourcing of the notebook's code to modules makes it easier to reuse it, and also makes the notebooks more readable.

However, for the code to work, `nominatim.py` needs to be in the same folder as a Jupyter notebook. If you want to call this module from another location, the path specification in the import would have to be changed. In this case it is better to create your own package, as described in [Packing](#).

3.6 Overview

3.6.1 Remote storage media

boto3

S3

azure-storage-blob

Azure

pydrive2

Google Drive

paramiko

SSH

PyArrow

HDFS

3.6.2 Geodata

Rasterio

reads and writes GeoTIFF and other forms of raster datasets.

Geospatial Data Abstraction Library (GDAL)

provides a low-level but more powerful API for reading and writing hundreds of data formats.

satpy

Easy to use API for sensors of satellite images like [MODIS](#), [Sentinel-2](#) etc.

sentinelSAT

Find and download Copernicus Sentinel satellite imagery using command line or Python.

fiona

reads and writes *.shp- and *.json data and many other formats.

pyproj

Python interface to [PROJ](#), a library for cartographic projections and coordinate transformations.

pyModis

Collection of Python scripts for downloading and mosaicking MODIS data.

Arcpy

is used by Esri ArcGIS to perform geographic data analysis, data conversion, data management, and map automation.

RSGISLib

or *The Remote Sensing and GIS Software Library* is a set of remote sensing tools for raster processing and analysis.

pgeocode

is used for querying of GPS coordinates and municipality names from postal codes, distances between postal codes as well as general distances.

3.7 Geodata

Rasterio

reads and writes GeoTIFF and other forms of raster datasets.

Geospatial Data Abstraction Library (GDAL)

provides a low-level but more powerful API for reading and writing hundreds of data formats.

satpy

Easy to use API for sensors of satellite images like [MODIS](#), [Sentinel-2](#) etc.

sentinelSAT

Find and download Copernicus Sentinel satellite imagery using command line or Python.

fiona

reads and writes *.shp- and *.json data and many other formats.

pyproj

Python interface to [PROJ](#), a library for cartographic projections and coordinate transformations.

pyModis

Collection of Python scripts for downloading and mosaicking MODIS data.

Arcpy

is used by Esri ArcGIS to perform geographic data analysis, data conversion, data management, and map automation.

RSGISLib

or *The Remote Sensing and GIS Software Library* is a set of remote sensing tools for raster processing and analysis.

pgeocode

is used for querying of GPS coordinates and municipality names from postal codes, distances between postal codes as well as general distances.

3.8 PostgreSQL

3.8.1 Basic functions

ACID compliant

ACID (**A** tomicity, **C** onsistency, **I** solation, **D** urability) is a series of properties that database transactions should fulfil to guarantee the validity of the data even in the event of a fault.

SQL:2011

[temporal_tables](#) also meet the SQL standard ISO/IEC 9075:2011, including:

- Time period definitions
- Valid time tables
- Transaction time tables (system-versioned tables) with time-sliced and sequenced queries

Data types

The following data types are supported out of the box:

- primitive data types: Integer, Numeric, String, Boolean
- structured data types: Date/Time, Array, Range, UUID
- document types: JSON/JSONB, XML, key-value ([Hstore](#))
- geometric data types: point, line, circle, polygon
- adjustments: composite, custom Types
- transactional data definition language (DDL)

Transactional DDL is implemented via [write-ahead logging](#). Big changes are also possible, but not adding and dropping databases and tables:

```
$ psql mydb
mydb=# DROP TABLE IF EXISTS foo;
NOTICE: table "foo" does not exist
DROP TABLE
mydb=# BEGIN;
BEGIN
mydb=# CREATE TABLE foo (bar int);
CREATE TABLE
mydb=# INSERT INTO foo VALUES (1);
INSERT 0 1
mydb=# ROLLBACK;
```

(continues on next page)

(continued from previous page)

```
ROLLBACK
mydb=# SELECT * FROM foo;
ERROR: relation "foo" does not exist
```

Concurrent Index

PostgreSQL can create indexes without having to lock write access to tables.

See also:

[Building Indexes Concurrently](#)

Extensions

PostgreSQL can easily be extended. The [contrib/](#) directory supplied with the source code contains various extensions that are described in [Appendix F](#). Other extensions have been developed independently, such as *PostGIS* or *Slony-I*.

Common Table Expression

[WITH Queries \(Common Table Expressions\)](#) divides complex queries into simpler queries, e.g.:

```
WITH regional_insolation AS (
    SELECT region, SUM(amount) AS total_insolation
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_insolation
    WHERE total_insolation > (SELECT SUM(total_insolation)/10 FROM regional_
↪insolation)
)
```

There is also a `RECURSIVE` modifier that refers the `WITH` query to its own output. The following is an example of how to sum the numbers from 1 to 100:

```
WITH RECURSIVE t (n) AS (
    WERTE (1)
    UNION ALL
    SELECT n + 1 FROM t WO <100
)
SELECT sum (n) FROM t;
```

Multi-Version Concurrency Control (MVCC)

[Multi-Version Concurrency Control](#) allows two or more sessions to access the same data at the same time without compromising the integrity of the data.

Cross platform

PostgreSQL runs on common CPU architectures such as x86, PowerPC, Sparc, ARM, MIPS or PA-RISC. Most operating systems are also supported: Linux, Windows, FreeBSD, OpenBSD, NetBSD, Mac OS, AIX, HP/UX and Solaris.

See also:

explain.depesz.com

Web app that visualises PostgreSQL's [EXPLAIN](#) and [ANALYZE](#) statements.

Foreign Data Wrappers (FDW)

In 2003, SQL was expanded to include SQL/MED (SQL Management of External Data). PostgreSQL 9.1 supports this read-only, 9.3 then also write. Since then, a number of Foreign Data Wrappers (FDW) have been developed for PostgreSQL.

The following is just a small selection of the best-known FDWs:

Note: Most of these wrappers are not officially supported by the PostgreSQL Global Development Group (PGDG).

Generic SQL wrappers

ODBC

Native ODBC FDW for PostgreSQL 9.5

- [GitHub](#)

Multicorn

[Multicorn](#) makes it easy to develop FDWs. For example, [SQLAlchemy](#) uses Multicorn to save your data in PostgreSQL.

- [GitHub](#)
- [PGXN](#)
- [Docs](#)

VirtDB

Native access to VirtDB (SAP ERP, Oracle RDBMS)

- [GitHub](#)

Specific SQL wrappers

postgres_fdw

With [postgres_fdw](#) data from other PostgreSQL servers can be accessed.

- [Git](#)
- [PGXN](#)
- [Docs](#)

Oracle

FDW for Oracle databases

- [GitHub](#)
- [PGXN](#)
- [Docs](#)

MySQL

FDW for MySQL from PostgreSQL9.3

- [GitHub](#)
- [PGXN](#)

SQLite

FDW for SQLite3

- [GitHub](#)
- [PGXN](#)
- [Docs](#)

NoSQL database wrappers

Cassandra

FDW für Cassandra

- [GitHub](#)
- [rankactive](#)

Neo4j

FWD for [Neo4j](#), which also provides a cypher function for PostgreSQL

- [GitHub](#)
- [Docs](#)

Redis

FDW for [Redis](#)

- [GitHub](#)

Riak

FDW for [Riak](#)

- [GitHub](#)

File wrappers

CSV

Official extension for PostgreSQL 9.1

- [Git](#)
- [Docs](#)

JSON

FDW for JSON files

- [GitHub](#)
- [Example](#)

XML

FDW for XML files

- [GitHub](#)
- [PGXN](#)

Geo wrappers

GDAL/OGR

FDW for the [GDAL/OGR](#) driver including databases like Oracle and SQLite as well as file formats like MapInfo, CSV, Excel, OpenOffice, OpenStreetMap PBF and XML.

- [GitHub](#)

Geocode/GeoJSON

A collection of FDWs for PostGIS

- [GitHub](#)

Open Street Map PBF

FDW for Open Street Map PBF

- [GitHub](#)

Generic web wrappers

ICAL

FDW for ICAL

- [GitHub](#)
- [Docs](#)

IMAP

FDW for the Internet Message Access Protocol (IMAP)

- [Docs](#)

RSS

FDQ for RSS feeds

- [Docs](#)

See also:

- [PostgreSQL wiki](#)
- [PGXN website](#)

Procedural programming languages

With PostgreSQL, user-defined functions can be written in languages other than SQL and C.

There are currently four procedural languages available in the standard PostgreSQL distribution:

- [PL/pgSQL](#)
- [PL/Tcl](#)
- [PL/Perl](#)
- [PL/Python](#)

Additional procedural programming languages are available but are not included in the core distribution:

- [PL/Java](#)
- [PL/Lua](#)

- [PL/R](#)
- [PL/sh](#)
- [PL/v8](#)

See also:

[External Procedural Languages](#)

In addition, other languages can be defined, see also [Writing A Procedural Language Handler](#).

DB-API 2.0

The Python API for database connectors is easy to use and understand. The two main concepts are:

Connection

[Connection Objects](#) allow the following methods:

connect(parameters...)

opens the connection to the database

.close()

closes the connection to the database

.commit()

transfers the outstanding transaction to the database

.rollback()

This method is optional as not all databases allow transactions to be rolled back.

.cursor ()

Return of a new cursor object via the connection.

Example:

```
import driver

conn = driver.connect(
    database="example",
    host="localhost",
    port=5432
)

try:
    # create the cursor
    # use the cursor
except Exception:
    conn.rollback()
else:
    conn.commit()
    conn.close()
```

Cursor

[Cursor objects](#) are used to manage the context of a `.fetch*()` method.

Cursors that are created in the same connection are not isolated from one another.

There are two attributes for cursor objects:

.description

contains the following seven elements:

1. name
2. type_code
3. display_size
4. internal_size
5. precision
6. scale
7. null_ok

The first two elements (name and type_code) are mandatory, the other five are optional and are set to None if no meaningful values can be specified.

.rowcount

indicates the number of lines that the last call of .execute*() with SELECT, UPDATE or INSERT resulted in.

Example:

```
cursor = conn.cursor()
cursor.execute(
    """
    SELECT column1, column2
    FROM tableA
    """
)
for column1, column2 in cursor.fetchall():
    print(column1, column2)
```

See also:

PEP 249 – Python Database API Specification v2.0

Psycopg

Psycopg is a PostgreSQL adapter based on the C library for PostgreSQL [libpq](#). Among other things, it offers:

- DB API 2.0 compatibility
- Multithreading with thread safety
- [Connections pooling](#) to be able to use a cache of existing database connections for queries.
- [Asynchronous](#) and [Coroutines](#) support
- [Adaptation of the Python types in SQL](#)

Install

With Spack you can provide `psycopg2` in your kernel, e.g. with

```
$ spack env activate python-311
$ spack install py-psycopg2
```

Object-relational mapping

«Object-relational mapping (...) in computer science is a programming technique for converting data between incompatible type systems using object-oriented programming languages.»¹

In the simplest case, classes are mapped to tables, with each object corresponding to a table row and each attribute to a table column.

There are essentially three different methods of mapping inheritance hierarchies:

Single Table

One table is created for each inheritance hierarchy, with all attributes of the base class and all classes derived from it being stored in a common table.

Joined Table or Class Table

A table is created for each subclass and a further table for each subclass derived from it.

Table per Class or Concrete Table

The attributes of the abstract base class are included in the tables for the specific subclasses. However, it is not possible to determine instances of different classes with one query.

SQLAlchemy

`SQLAlchemy` is a Python-SQL-Toolkit and object-relational mapper.

`SQLAlchemy` is known for its ORM, whereby it provides different patterns for object-relational mapping, whereby classes can be mapped to the database in different ways. The object model and the database schema are cleanly decoupled from the start.

`SQLAlchemy` differs fundamentally from other ORMs, as SQL and details of the object relation are not abstracted away: all processes are represented as a collection of individual tools.

`SQLAlchemy` supports PostgreSQL as well as other dialects of relational databases:

Dialects	Python package	import	Docs
postgresql	psycopg2-binary	psycopg2	Installation
mysql	mysqlclient	MySQLdb	README
mssql	pyodbc	pyodbc	Wiki
oracle	cx_oracle	cx_Oracle	cx_Oracle

¹ Wikipedia: relational mapping

Database connection

```
from sqlalchemy import create_engine

engine = create_engine("postgresql:///example", echo=True)
```

Data model

```
from sqlalchemy import Column, ForeignKey, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class Address(Base):
    __tablename__ = "address"

    id = Column(Integer, primary_key=True)
    street = Column(String)
    zipcode = Column(String)
    country = Column(String, nullable=False)

class Contact(Base):
    __tablename__ = "contact"

    id = Column(Integer, primary_key=True)

    firstname = Column(String, nullable=False)
    lastname = Column(String, nullable=False)
    email = Column(String, nullable=False)
    address_id = Column(Integer, ForeignKey(Address.id), nullable=False)
    address = relationship("Address")
```

Create tables

```
Base.metadata.create_all(engine)
```

Create Session

```
session = Session(engine)
address = Address(street="Birnbaumweg 10", zipcode="79115", country="Germany")

contact = Contact(
    firstname="Veit", lastname="Schiele", email="veit@cusy.io", address=address
)

session.add(contact)
session.commit()
```

Read

```
contact = session.query(Contact).filter_by(email="veit@cusy.io").first()
print(contact.firstname)

contacts = session.query(Contact).all()
for contact in contacts:
    print(contact.firstname)

contacts = session.query(Contact).filter_by(email="veit@cusy.io").all()
for contact in contacts:
    print(contact.firstname)
```

Update

```
contact = session.query(Contact).filter_by(email="veit@cusy.io").first()
contact.email = "info@veit-schiele.de"

session.add(contact)
session.commit()
```

Delete

```
contact = (
    session.query(Contact).filter_by(email="info@veit-schiele.de").first()
)

session.delete(contact)
session.commit()
```


Extensions

SQLAlchemy-Continuum

Versioning and revision extension for SQLAlchemy

SQLAlchemy-Utc

SQLAlchemy type for storing *datetime.datetime* values

SQLAlchemy-Utils

Various utility functions, new data types and utilities for SQLAlchemy

DEPOT

Framework for easy storage and retrieval of files in web applications

SQLAlchemy-ImageAttach

SQLAlchemy extension for attaching images to entity objects

SQLAlchemy-Searchable

Full-text searchable models for SQLAlchemy

See also:

- [Awesome SQLAlchemy](#)

Alembic

Alembic is based on SQLAlchemy and serves as a database migration tool with the following functions:

- ALTER statements to a database to change the structure of tables and other constructs
- System for creating migration scripts. Optionally, the sequence of steps for the downgrade can also be specified.
- The scripts are executed in a specific order.

See also:

[Auto Generating Migrations](#)

Create migration environment

The Migration Environment is a directory that is specific to a particular application. It is created with the Alembic `ini` command and then managed along with the application's source code.

```
$ cd myproject
$ alembic init alembic
Creating directory /path/to/myproject/alembic...done
Creating directory /path/to/myproject/alembic/versions...done
Generating /path/to/myproject/alembic.ini...done
Generating /path/to/myproject/alembic/env.py...done
Generating /path/to/myproject/alembic/README...done
Generating /path/to/myproject/alembic/script.py.mako...done
Please edit configuration/connection/logging settings in
'/path/to/myproject/alembic.ini' before proceeding.
```

The structure of such a migration environment can for example look like this:

```
myproject/
├── alembic
│   ├── alembic.ini
│   ├── env.py
│   ├── README
│   ├── script.py.mako
│   └── versions
│       ├── 2b1ae634e5cd_add_order_id.py
│       ├── 3512b954651e_add_account.py
│       └── 3adcc9a56557_rename_username_field.py
```

Templates

Alembic includes a number of templates that can be displayed with list:

```
$ alembic list_templates
Available templates:
```

```
generic - Generic single-database configuration.
multidb - Rudimentary multi-database configuration.
pylons - Configuration that reads from a Pylons project environment.
```

Templates are used via the 'init' command, e.g.:

```
alembic init --template pylons ./scripts
```

Configure ini file

The file created with the generic template looks like this:

```
# A generic, single database configuration.

[alembic]
# path to migration scripts
script_location = alembic

# template used to generate migration files
# file_template = %(rev)s_%(slug)s

# timezone to use when rendering the date
# within the migration file as well as the filename.
# string value is passed to dateutil.tz.gettz()
# leave blank for localtime
# timezone =

# max length of characters to apply to the
# "slug" field
#truncate_slug_length = 40

# set to 'true' to run the environment during
```

(continues on next page)

(continued from previous page)

```

# the 'revision' command, regardless of autogenerate
# revision_environment = false

# set to 'true' to allow .pyc and .pyo files without
# a source .py file to be detected as revisions in the
# versions/ directory
# sourceless = false

# version location specification; this defaults
# to alembic/versions.  When using multiple version
# directories, initial revisions must be specified with --version-path
# version_locations = %(here)s/bar %(here)s/bat alembic/versions

# the output encoding used when revision files
# are written from script.py.mako
# output_encoding = utf-8

sqlalchemy.url = driver://user:pass@localhost/dbname

# Logging configuration
[loggers]
keys = root,sqlalchemy,alembic

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = WARN
handlers = console
qualname =

[logger_sqlalchemy]
level = WARN
handlers =
qualname = sqlalchemy.engine

[logger_alembic]
level = INFO
handlers =
qualname = alembic

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic

[formatter_generic]
format = %(levelname)-5.5s [% (name)s] %(message)s

```

(continues on next page)

(continued from previous page)

```
datefmt = %H:%M:%S
```

%(here)s

Replacement variable for creating absolute paths

file_template

This is the naming scheme used to generate new migration files. The available variables include:

%(rev)s

Revision ID

%(slug)s

Abbreviated revision message

%(year)d, %(month).2d, %(day).2d, %(hour).2d, %(minute).2d, %(second).2d

Creation time

Create a migration script

A new revision can be created with:

```
$ alembic revision -m "create account table"
Generating /path/to/yourproject/alembic/versions/1975ea83b712_create_account_table.py...
↪ done
```

Then the file `1975ea83b712_create_account_table.py` looks like this:

```
"""create account table

Revision ID: 1975ea83b712
Revises:
Create Date: 2018-12-08 11:40:27.089406

"""

# revision identifiers, used by Alembic.
revision = "1975ea83b712"
down_revision = None
branch_labels = None

import sqlalchemy as sa

from alembic import op

def upgrade():
    pass

def downgrade():
    pass
```

down_revision

Variable that tells Alembic in which order the migrations should be carried out, for example:

```
# revision identifiers, used by Alembic.
revision = "ae1027a6acf"
down_revision = "1975ea83b712"
```

upgrade, downgrade

for example:

```
def upgrade():
    op.create_table(
        "account",
        sa.Column("id", sa.Integer, primary_key=True),
        sa.Column("name", sa.String(50), nullable=False),
        sa.Column("description", sa.Unicode(200)),
    )

def downgrade():
    op.drop_table("account")
```

`create_table()` and `drop_table()` are Alembic directives. You can get an overview of all Alembic directives in the [Operation Reference](#).

Run migration

First migration:

```
$ alembic upgrade head
INFO [alembic.context] Context class PostgresqlContext.
INFO [alembic.context] Will assume transactional DDL.
INFO [alembic.context] Running upgrade None -> 1975ea83b712
```

We can also refer directly to revision numbers:

```
$ alembic upgrade ae1
```

Relative migrations can also be initiated:

```
$ alembic upgrade +2
```

or:

```
$ alembic downgrade -1
```

or:

```
$ alembic upgrade ae10+2
```

Display Information

Current version

```
$ alembic current
INFO [alembic.context] Context class PostgresqlContext.
INFO [alembic.context] Will assume transactional DDL.
Current revision for postgresql://scott:XXXXX@localhost/test: 1975ea83b712 ->
↪ ae1027a6acf (head), Add a column
```

History

```
$ alembic history --verbose

Rev: ae1027a6acf (head)
Parent: 1975ea83b712
Path: /path/to/yourproject/alembic/versions/ae1027a6acf_add_a_column.py

    add a column

Revision ID: ae1027a6acf
Revises: 1975ea83b712
Create Date: 2014-11-20 13:02:54.849677

Rev: 1975ea83b712
Parent: <base>
Path: /path/to/yourproject/alembic/versions/1975ea83b712_add_account_table.py

    create account table

Revision ID: 1975ea83b712
Revises:
Create Date: 2014-11-20 13:02:46.257104
```

The history can also be displayed more specifically:

```
$ alembic history -r1975ea:ae1027
```

or:

```
$ alembic history -r-3:current
```

or:

```
$ alembic history -r1975ea:
```

ipython-sql

`ipython-sql` introduces the `%sql` or `%%sql` magics for `iPython` and `Jupyter` notebooks.

Installation

You can easily install `ipython-sql` in your `Jupyter` kernel with:

```
$ pipenv install ipython-sql
```

First steps

1. First, `ipython-sql` is activated in your notebook with

```
In [1]: %load_ext sql
```

2. The [SQLAlchemy URL](#) is used to connect to the database:

```
In [2]: %sql postgresql://
```

3. Then you can create a table, for example:

```
In [3]: %%sql postgresql://
....: CREATE TABLE accounts (login, name, email)
....: INSERT INTO accounts VALUES ('veit', 'Veit Schiele', veit@example.org);
```

4. You can query the contents of the `accounts` table with

```
In [4]: result = %sql select * from accounts
```

Configuration

Query results are loaded as a list, so very large amounts of data can occupy memory. Usually there is no automatic limit, but with `Autolimit` you can limit the amount of results.

Note: `displaylimit` only limits the amount of results displayed, but not the amount of memory required.

With `%config SqlMagic` you can display the current configuration:

```
In [4]: %config SqlMagic
SqlMagic options
-----
SqlMagic.autocommit=<Bool>
  Current: True
  Set autocommit mode
SqlMagic.autolimit=<Int>
  Current: 0
  Automatically limit the size of the returned result sets
SqlMagic.autopandas=<Bool>
```

(continues on next page)

(continued from previous page)

```
Current: False
Return Pandas DataFrames instead of regular result sets
...
```

Note: If autopandas is set to True, `displaylimit` is not applied. In this case, the `max_rows` option of pandas can be used as described in the [pandas documentation](#).

pandas

If pandas is installed, the DataFrame method can be used:

```
In [5]: result = %sql SELECT * FROM accounts

In [6]: dataframe = result.DataFrame()

In [7]: %sql --persist dataframe

In [8]: %sql SELECT * FROM dataframe;
```

--persist

Argument with the name of a DataFrame object, creates a table name in the database from this.

--append

Argument to add rows with this name to an existing table.

PostgreSQL features

Meta-commands from `psql` can also be used in `ipython-sql`:

-l, --connections

lists all active connections

-x, --close *SESSION-NAME*

close named connection

-c, --creator *CREATOR-FUNCTION*

specifies the creator function for a new connection

-s, --section *SECTION-NAME*

specifies section of `dsn_file` to be used in a connection

-p, --persist

creates a table in the database from a named DataFrame

--append

similar to `--persist`, but the contents are appended to the table

-a, --connection_arguments "*{CONNECTION-ARGUMENTS}*"

specifies a dict of connection arguments to be passed to the SQL driver

-f, --file *PATH*

executes SQL from the file under this path

See also:

- [pgspecial](#)

Warning: Since `ipython-sql` processes `--` options such as `--persist`, and at the same time accepts `--` as a SQL comment, the parser has to make some assumptions: for example, `--persist is great` in the first line is processed as an argument and not as a comment.

PostGIS

PostGIS is an extension for PostgreSQL that includes geographic objects and functions. The extension implements i.a. the [Simple Feature Access](#) specification of the [Open Geospatial Consortium](#). Although PostgreSQL already supports geometry types, these are insufficient for geographic tasks. Therefore, PostGIS creates its own data types that are better suited for geographic tasks. The following geometry types are supported:

- OpenGIS with well-known text and well-known binary
- Extended Well-Known Text and Extended Well-Known Binary also with height information and/or measured values
- SQL/MM with Circularstring, Compoundcurve, Curvepolygon, Multicurve and Multisurface

[GEOS](#), on the other hand, contains the numerous spatial functions and operators for geographic data.

Finally, [pgRouting](#) contains routing functions based on PostGIS.

In the [OpenStreetMap](#) project, PostGIS is used to render maps with [Mapnik](#).

Install PostGIS

For Ubuntu 22.04 you can simply install PostGIS with:

```
$ sudo apt install postgis
```

Then you can activate PostGIS.

1. Switch to the PostgreSQL user:

```
$ sudo -i -u postgres
```

2. Create test user and database:

```
$ createuser postgis
$ createdb postgis_db -O postgis
```

3. Establish a connection to the database:

```
$ psql -d postgis_db
psql (14.5 (Ubuntu 14.5-0ubuntu0.22.04.1))
Type "help" for help.
```

4. Activate the PostGIS extension in the database:

```
ppostgis_db = # CREATE EXTENSION postgis;
CREATE EXTENSION
```

5. Check that PostGIS is working:

```
postgis_db=# SELECT PostGIS_version();
           postgis_version
-----
3.2 USE_GEOS=1 USE_PROJ=1 USE_STATS=1
(1 row)
```

See also:

- [PostGIS Installation](#)

Optimising PostgreSQL for GIS database objects

In the standard installation, PostgreSQL is configured very cautiously so that it can run on as many systems as possible. However, GIS database objects are large compared to text data. Therefore, PostgreSQL should be configured to work better with these objects. To do this, we configure the `/etc/postgresql/14/main/postgresql.conf` file as follows:

1. `shared_buffer` should be changed to approx. 75% of the total working memory, but never fall below 128 kB:

```
shared_buffers = 768MB
```

2. `work_mem` should be increased to at least 16MB:

```
work_mem = 16MB
```

3. `maintenance_work_mem` should be increased to 128MB:

```
maintenance_work_mem = 128MB
```

4. Finally, `random_page_cost` should be set to 2.0.

```
random_page_cost = 2.0
```

PostgreSQL should be restarted for the changes to take effect:

```
$ sudo service postgresql restart
```

Loading geospatial data

Now let's load some geospatial data into our database so that we can familiarise ourselves with the tools and processes used to retrieve that data.

[Natural Earth](#) provides a great source of basic data for the whole world on various scales. And the best thing is that this data is in the public domain:

1. Download the data

```
$ mkdir nedata
$ cd !$
cd nedata
$ wget https://www.naturalearthdata.com/http://www.naturalearthdata.com/download/
↪ 110m/cultural/ne_110m_admin_0_countries.zip
```

2. Unzip the file

```
$ sudo apt install unzip
$ unzip ne_110m_admin_0_countries.zip
Archive: ne_110m_admin_0_countries.zip
  inflating: ne_110m_admin_0_countries.README.html
  extracting: ne_110m_admin_0_countries.VERSION.txt
  extracting: ne_110m_admin_0_countries.cpg
  inflating: ne_110m_admin_0_countries.dbf
  inflating: ne_110m_admin_0_countries.prj
  inflating: ne_110m_admin_0_countries.shp
  inflating: ne_110m_admin_0_countries.shx
```

3. Load into our postgis_db database

The files `.dbf`, `.prj`, `.shp` and `.shp` form a so-called ShapeFile, a popular geospatial data format that is used by GIS software. To load this into our database, we also need [GDAL](#), the *Geospatial Data Abstraction Library*. When we install GDAL we also get OGR, *OpenGIS Simple Features Reference Implementation*, a vector data translation library that we can use to translate the shapefile into data.

1. GDAL can be easily installed with the package manager:

```
$ sudo apt install gdal-bin
```

2. Then we switch to the postgresql user:

```
$ sudo -i -u postgres
```

3. Now we convert the shapefile with ogr2ogr and import it into our database:

```
$ ogr2ogr -f PostgreSQL PG:dbname=postgis_db -progress \
  -nlt PROMOTE_TO_MULTII \
  /srv/jupyter/nedata/ne_110m_admin_0_countries.shp
0...10...20...30...40...50...60...70...80...90...100 - done.
```

-f PostgreSQL

indicates that the target is a PostgreSQL database

PG:dbname=postgis_db

specifies the PostgreSQL database name. In addition to the name, other options can also be specified, in general:

```
PG:"dbname='db_ename' host='addr' port='5432' user='x' password='y'"
```

-progress

outputs a progress bar

-nlt PROMOTE_TO_MULTII

indicates that all object types should be loaded into the database as multipolygons

/home/veit/nedata/ne_110m_admin_0_countries.shp

specifies the path to the input file

See also:

- [ogr2ogr](#)

4. Check the import with ogrinfo

```
$ ogrinfo -so PG:dbname=postgis_db ne_110m_admin_0_countries
Output
INFO: Open of `PG:dbname=postgis_db'
      using driver `PostgreSQL' successful.

Layer name: ne_110m_admin_0_countries
Geometry: Multi Polygon
Feature Count: 177
...
```

5. Alternatively, we can also list individual tables:

```
$ psql -d postgis_db
postgis_db=# \dt
              List of relations
 Schema |          Name          | Type  | Owner
-----+-----+-----+-----
 public | ne_110m_admin_0_countries | table | postgres
 public | spatial_ref_sys         | table | postgres
(2 rows)
```

6. Finally, we can log out of the database with

```
psql> \q
```

See also:

- [PostGIS Reference](#)

Database security

Database permissions

The PostgreSQL login via superuser `postgres` should only ever be allowed via Unix domain sockets and via `localhost`. Access with [peer authentication](#) in the `pg_hba.conf`, however, can be granted:

#	TYPE	DATABASE	USER	ADDRESS	METHOD
local		all	postgres		peer
host		all	all	10.23.42.1/24	scram-sha-256

The database should be created by the database administrator and then configured in such a way that not everyone (PUBLIC) can connect to it:

```
CREATE DATABASE myapp;
REVOKE ALL ON myapp FROM PUBLIC;
```

This means that only the superuser can connect to the myapp database.

Save passwords

Passwords should never be in plain text, e.g. also not be saved in an `.env` file. When saving and transmitting passwords, this should always be [salted](#). For PostgreSQL there is the extension [pgcrypto](#), which can be easily activated with

```
CREATE EXTENSION pgcrypto;
```

For this reason, secure passwords should be assigned when they are created, which can then get saved e.g. in [Vault](#) or similar:

```
CREATE ROLE myapp_users;
CREATE ROLE myapp_reader IN ROLE myapp_users LOGIN PASSWORD '...';
CREATE ROLE myapp_writer IN ROLE myapp_users LOGIN PASSWORD '...';
```

Then users with the role `myapp_users` first get `CONNECT` rights and then `myapp_reader` read rights and `myapp_writer` write rights:

```
GRANT CONNECT ON DATABASE to myapp_users;
GRANT SELECT ON diagnosis_key TO myapp_reader;
GRANT INSERT ON diagnosis_key TO myapp_writer;
```

The user `myapp_reader` can, however, read all data at once. This is also a point of attack that is better cut by a function:

```
CREATE OR REPLACE FUNCTION get_key_data(in_id UUID)
  RETURNS JSONB
  AS 'SELECT key_data FROM diagnosis_key WHERE id = in_id;'
  LANGUAGE sql SECURITY DEFINER SET search_path = :schema, pg_temp;
```

Then the function `myapp_owner` is assigned, the authorisations for `myapp_reader` and `myapp_writer` are revoked and finally the execution of the function `myapp_reader` is allowed:

```
ALTER FUNCTION get_key_data(UUID) OWNER TO myapp_owner;
REVOKE ALL ON FUNCTION get_key_data(UUID) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION get_key_data(UUID) TO myapp_reader;
```

This means that `myapp_reader` can only read a single data record.

id

the `id` shouldn't be written as `serial`, `bigserial` or similar. Counting numbers could be easily guessed by attackers. Therefore the `UUIDv4` data type is much more suitable. In PostgreSQL you can generate `UUIDv4` with the [uuid-oss](#) extension or for PostgreSQL9.4 also the [pgcrypto](#) extension:

```
CREATE EXTENSION "uuid-oss";
CREATE TABLE diagnosis_key (
  id uuid primary key default uuid_generate_v4() NOT NULL,
  ...
);
```

or

```
CREATE EXTENSION "pgcrypto";
CREATE TABLE diagnosis_key (
```

(continues on next page)

(continued from previous page)

```
id uuid primary key default gen_random_uuid() NOT NULL,  
...  
);
```

Time stamp

Occasionally, the date and time are stored as `bigint`, i.e. as a number, even though there is also a `TIMESTAMP` data type. This would have the advantage that you can easily count on them, for example:

```
SELECT age(submission_timestamp);  
SELECT submission_timestamp - '1 day'::interval;
```

In addition, the data could be deleted after a certain period of time, e.g. after thirty days with:

```
DELETE FROM diagnosis_key WHERE age(submission_timestamp) > 30;
```

Deletion can be accelerated if a separate partition is created for each day with the PostgreSQL extension `pg_partman`.

See also:

- [Veil2 – Relational Security for Postgres](#)
- [PostgreSQL Secure Monitoring \(Posemo\)](#)

PostgreSQL performance

You shouldn't start with *MVCC – Multiversion Concurrency Control* if you want to optimise your PostgreSQL database: many improvements can be made much easier since neither transaction logs nor large Linux kernel page sizes are likely to be responsible. Usually we start with two metrics that can very well indicate the performance of your databases:

Cache and index hit rate

Cache hit ratio

Percentage of time that data can be served from RAM instead of hard disk space. For a web app with many small requests, I recommend about 99%.

```
SELECT  
  'index hit rate' AS name,  
  (sum(idx_blks_hit)) / nullif(sum(idx_blks_hit + idx_blks_read),0) AS ratio  
FROM pg_statio_user_indexes  
UNION ALL  
SELECT  
  'table hit rate' AS name,  
  sum(heap_blks_hit) / nullif(sum(heap_blks_hit) + sum(heap_blks_read),0) AS ratio  
FROM pg_statio_user_tables;
```

If the cache hit rate is too low, you can simply increase the memory.

Index hit ratio

Frequency of use of the indices.

```

SELECT relname,
       CASE idx_scan
         WHEN 0 THEN 'Insufficient data'
         ELSE (100 * idx_scan / (seq_scan + idx_scan))::text
       END percent_of_times_index_used,
       n_live_tup rows_in_table
FROM   pg_stat_user_tables
ORDER BY
       n_live_tup DESC;

```

relname	percent_of_times_index_used	rows_in_table
account	11	5409
activity	69	58276
application	93	5345
...		

Typically, we shouldn't have more than 10,000 records in a table and the percentage of the index used should be greater than 90%.

In our example, we see that the `account` table is missing relevant indices, as an index is only used in 11% of the queries. The `activity` table is also missing some suitable indices, but it also has a lot of records, so it might make sense to split it into several tables.

Clean up unused indices

Unused indices lead to a slower throughput when writing the data sets without making queries faster.

```

SELECT
  schemaname || '.' || relname AS table,
  indexrelname AS index,
  pg_size_pretty(pg_relation_size(i.indexrelid)) AS index_size,
  idx_scan AS index_scans
FROM   pg_stat_user_indexes ui
JOIN   pg_index i ON ui.indexrelid = i.indexrelid
WHERE  NOT indisunique AND idx_scan < 50 AND pg_relation_size(relid) > 5 * 8192
ORDER BY pg_relation_size(i.indexrelid) / nullif(idx_scan, 0) DESC NULLS FIRST,
         pg_relation_size(i.indexrelid) DESC;

```

Indices that are not used can simply be removed. On the other hand the decision becomes more difficult for indices that are only used very rarely: here a trade-off must be made between the write and the query speed.

Clean up unused data

Although PostgreSQL can hold a wide variety of data, it is not always useful to do so. Tables such as `messages`, `logs` and `events` have a good chance of taking up most of the memory without directly benefiting the database application: if this data is rather for monitoring or error analysis, it should be stored outside the database and rotated regularly.

Analyse query performance with pg_stat_statements

`pg_stat_statements` records queries and keeps a number of statistics on them. Thus, at regular intervals, we check which queries are the slowest on average and which put the greatest load on the system:

```
SELECT
  (total_time / 1000 / 60) as total_minutes,
  (total_time/calls) as average_time,
  query
FROM pg_stat_statements
ORDER BY 1 DESC
LIMIT 50;
```

total_time	avg_time	query
295.761165833319	10.1374053278061	SELECT id FROM account WHERE email LIKE ?
219.138564283326	80.24530822355305	SELECT * FROM account WHERE user_id = ? AND
current = True		
...		

Typical response times should be ~1ms and in a few cases ~4-5ms. To start optimising performance, we usually weigh the total time against the average time, so in the above example we would probably start with the second line as we see the greater potential for savings here. To get a more accurate idea of the query, we analyse it more closely with:

```
EXPLAIN ANALYZE
SELECT *
FROM account
WHERE user_id = 123
      AND current = True
```

QUERY PLAN

```
Aggregate (cost=4690.88..4690.88 rows=1 width=0) (actual time=519.288..519.289 rows=1
loops=1)
-> Nested Loop (cost=0.00..4690.66 rows=433 width=0) (actual time=15.302..519.076
rows=213 loops=1)
-> Index Scan using idx_account_userid on account (cost=0.00..232.52 rows=23
width=4) (actual time=10.143..62.822 rows=1 loops=8)
Index Cond: (user_id = 123)
Filter: current
Rows Removed by Filter: 14
Total runtime: 219.428 ms
(1 rows)
```

So we see that although an index is used, 15 different rows are retrieved from it, of which 14 are then discarded. To optimise this, we would create a conditional or a composite index. In the first case `current = true` would have to be met, in the second case a composite index would be created with both values. A conditional index is usually more useful with a small set of values, while the composite index is more beneficial with larger sets of values. In our example, a conditional index clearly makes more sense. We can create this with:

```
CREATE INDEX CONCURRENTLY idx_account_userid_current ON account(user_id) WHERE current =
True;
```

Now the query plan should also improve:


```
EXPLAIN ANALYZE
SELECT *
FROM account
WHERE user_id = 123
      AND current = True
```

QUERY PLAN

```
-----
↪ Aggregate (cost=4690.88..4690.88 rows=1 width=0) (actual time=519.288..519.289 rows=1 loops=1)
↪   -> Index Scan using idx_account_userid_current on account (cost=0.00..232.52 rows=23 width=4) (actual time=10.143..62.822 rows=1 loops=8)
↪       Index Cond: ((user_id = 123) AND (current = True))
Total runtime: .728 ms
(1 rows)
```

pgMonitor

[pgMonitor](#) is an environment to visualise the health and performance of a PostgreSQL cluster. It combines a suite of tools to facilitate the collection of important metrics, including:

- number of connections
- Database size
- Replication lag
- Transaction wraparound
- Extra space taken up by your tables and indexes
- CPU, memory, I/O and uptime

It combines multiple open-source software packages to create a robust PostgreSQL monitoring environment, including:

PostgreSQL Exporter

an open-source data export to Prometheus that supports collecting metrics from any PostgreSQL server 9.1.

Prometheus

an open-source metrics collector that is highly customisable.

Grafana

an open-source data visualiser that allows you to generate many different kinds of charts and graphs.

See also:

- [pgexporter](#)

Installation and configuration

Installation and configuration instructions for each package are provided:

1. [PostgreSQL Exporter](#)
2. [Prometheus](#)
3. [Grafana](#)

pganalyze

[pganalyze](#) analyses the query plans of PostgreSQL. Currently it collects information about

- schema with tables (columns, constraints, trigger definitions) and indices
- Statistics on tables indices, databases and queries
- Operating system (OS, RAM, storage)

See also:

- [GitHub](#)
- [Docs](#)

Installation

1. Create a monitoring user for pganalyze:

```
CREATE USER pganalyze WITH PASSWORD '...' CONNECTION LIMIT 5;
GRANT pg_monitor TO pganalyze;
CREATE SCHEMA pganalyze;
GRANT USAGE ON SCHEMA pganalyze TO pganalyze;
REVOKE ALL ON SCHEMA public FROM pganalyze;
CREATE OR REPLACE FUNCTION pganalyze.get_stat_replication() RETURNS SETOF pg_stat_
↪replication AS
$$ /* pganalyze-collector */ SELECT * FROM pg_catalog.pg_stat_replication;
$$ LANGUAGE sql VOLATILE SECURITY DEFINER;
```

2. Check the connection:

```
PGPASSWORD=... psql -h localhost -d mydb -U pganalyze
```

3. Activate the pg_stat_statements:

```
ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements';
```

4. Restart of the PostgreSQL daemon:

```
$ sudo service postgresql restart
```

5. Checking pg_stat_statements:

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;
SELECT calls, query FROM pg_stat_statements LIMIT 1;
calls | query
-----+-----
      8 | SELECT * FROM t WHERE field = ?
(1 row)
```

6. Installing the *Collector*:

```
$ curl -L https://packages.pganalyze.com/pganalyze_signing_key.asc | sudo apt-key_
↪add -
$ echo "deb [arch=amd64] https://packages.pganalyze.com/ubuntu/bionic/ stable main" _
↪| sudo tee /etc/apt/sources.list.d/pganalyze_collector.list
$ sudo apt-get update
$ sudo apt-get install pganalyze-collector
```

7. Creating the API key

For the next step you need the pganalyze `api_key`. You can create this at the site <https://app.pganalyze.com/>

8. Configure the *collector*:

```
[pganalyze]
api_key: ...

[server]
db_host: 127.0.0.1
db_port: 5432
db_name: postgres, *
db_username: pganalyze
db_password: ...
```

9. Testing the *Collector* configuration:

```
$ sudo pganalyze-collector --test --reload
```

See also:

- [Installation Guide](#)

Log analysis

In order to continuously monitor, classify and statistically evaluate the local log files, `db_log_location` must be specified in `pganalyze-collector.conf`. `pganalyze-collector` provides help to find the log files:

```
$ pganalyze-collector --discover-log-location
```

The output can then look like this, for example:

```
db_log_location = /var/log/postgresql/postgresql-12-main.log
```

After this result has been entered in the `pganalyze-collector.conf` configuration file you can test it with:

```
$ pganalyze-collector --test
```

The result can then look like this, for example:

```
2021/02/06 06:40:06 I [server1] Testing statistics collection...
2021/02/06 06:40:07 I [server1] Test submission successful (15.8 KB received)
2021/02/06 06:40:07 I [server1] Testing local log tailing...
2021/02/06 06:40:13 I [server1] Log test successful
2021/02/06 06:40:13 I Re-running log test with reduced privileges of "pganalyze" user.
↪(uid = 107, gid = 113)
2021/02/06 06:40:13 I [server1] Testing local log tailing...
2021/02/06 06:40:19 I [server1] Log test successful
```

If the test was successful, the *Collector* must be restarted for the configuration to take effect:

```
$ systemctl restart pganalyze-collector
```

3.9 NoSQL databases

So far there is no uniform definition of NoSQL, but most NoSQL database systems usually have the following in common:

- no relational data model
- distributed and horizontal scalability
- no or weak schema restrictions
- simple API
- no *ACID*, but *Eventual consistency* or *BASE* as the consistency model

NoSQL databases can be divided into

3.9.1 Key-value database systems

Key-value databases, also known as key value stores, store *key/value pairs*.

Database systems

Key/value database systems are e.g. Riak, Cassandra, Redis and MongoDB.

Home	Riak	Cassandra	Redis	MongoDB
GitHub	basho/riak	apache/cassandra	redis/redis	mongodb/mongo
Docs	docs.riak.com	cassandra.apache.org/doc/	redis.io/documentation	docs.mongodb.com
Application areas	Session storage, Log data, Sensor data, CMS	Georedudancy, high writing speed, democratic peer-to-peer (P2P) architecture, data with a defined life-time	Session Cache, Full Page Cache (FPC), Queues, Pub/Sub	IoT, Mobile apps, CMS, simple geospatial data, ...
Development language	Erlang	Java	ANSI C	C++
Licenses	Apache License 2.0	Apache License 2.0	Redis Source Available License v2, Server-Side Public License v1	Server Side Public License
Data model	Essentially <i>Key/value pair</i>	<i>Column Family</i> correspond to tables, keyspaces to databases; no logical structure, no scheme	Keys are stored as strings, values as strings, hashes, lists, sets and sorted sets	Flexible scheme with denormalised model
Query language	Keyfilter, <i>MapReduce</i> , Link walking, no ad hoc queries possible	<i>Cassandra Query Language (CQL)</i>		jQuery, <i>MapReduce</i>
Transactions, concurrency	<i>ACID</i>	<i>Eventual Consistency</i>	in-memory, asynchronous on disc with <i>Append Only File Mode</i>	<i>Two-phase locking (2PL)</i>
Replication, skaling	Multi-master replication	SimpleStrategy, NetworkTopologyStrategy and OldNetworkTopologyStrategy	Master-N-Slaves replication, Sharding using <i>Consistent hash function</i>	Master-Slave replication, Auto-Sharding
Remarks		See also <i>Scylla</i> , a Cassandra-compatible reimplementation in C.	See also: KeyDB a fork with multithreading Redict a fork, licenced under LGPL-3.0 Valkey a fork by the Linux Foundation	<i>BSON</i> with a maximum document size of 16 MB.

3.9.2 Column-oriented database systems

Column-oriented databases, also known as wide column stores, store data from several entries together with a time stamp in columns. Columns with similar or related content can be combined in a *Column family*.

Database systems

Examples of column-oriented database systems are *Cassandra*, *Hypertable* and *HBase*.

Home	Cassandra	Hypertable	HBase
GitHub	apache/cassandra	vicaya/hypertable	apache/hbase
Docs	cassandra.apache.org/doc/	hypertable.com/documentation	hbase.apache.org/book.html
Application areas	Georedundancy, high writing speed, democratic peer-to-peer (P2P) architecture, data with a defined lifetime	Hypertable's Bigtable design solves horizontal scaling problems through a distributed storage system for structured data.	IoT, fraud detection, recommendation engines
Development language	Java	C++	Java
Licenses	Apache License 2.0	GPL-3.0 License	Apache-2.0 License
Data model	<i>Column Family</i> correspond to tables, <i>Keyspaces</i> databases; no logical structure, no scheme	Associative arrays	Tables divided into regions
Query language	Cassandra Query Language (CQL)	Hypertable Query Language (HQL)	Java Client API, Thrift/REST API
Transactions, concurrency	<i>Eventual Consistency</i>	<i>MVCC – Multiversion Concurrency Control</i>	<i>ACID</i> per line, <i>MVCC – Multiversion Concurrency Control</i>
Replication, scaling	SimpleStrategy, NetworkTopologyStrategy and OldNetworkTopologyStrategy	File system level replication	Master-Slave-Replication
Remarks		is based on distributed file systems such as Apache Hadoop, DFS or GlusterFS	

3.9.3 Document-oriented database systems

A document in this context is a structured compilation of certain data. The data of a document is stored as a *Key/value pair*, whereby the value can also be a list or an array.

Database systems

Document-oriented database systems are, for example, MongoDB, CouchDB, Riak, OrientDB and ArangoDB.

Home	MongoDB	CouchDB	Riak	OrientDB	ArangoDB
GitHub	mongodb/mongo	apache/couchdb	basho/riak	orienttechnologies/orientdb	arangodb/arangodb
Docs	docs.mongodb.com	docs.couchdb.org	docs.riak.com	www.orientdb.com	arangodb.com/documentation/
Application areas	IoT, Mobile apps, simple geospatial data, ...	Mobile, CRM, CMS, ...	Session storage, Log data, Sensor data, CMS	Master data management, social networks, Time Series, Key Value, Chat, traffic management	Fraud Detection, IoT, identity management, e-commerce, network, logistics, CMS
Development language	C++	Erlang	Erlang	Java	C++, JavaScript
Licenses	Server Side Public License	Apache License 2.0	Apache License 2.0	Apache License 2.0	Apache License 2.0
Data model	Flexible scheme with denormalised model	Flexible scheme	Essentially <i>Key/Value pair</i>	Multi-Model	Multi-model: documents, graphs and <i>Key/value pair</i>
Query language	jQuery, <i>MapReduce</i>	REST, <i>MapReduce</i>	Key filter, <i>MapReduce</i> , link walking, no ad-hoc queries possible	Gremlin	ArangoDB Query Language (AQL)
Transactions, concurrency	<i>Two-phase locking (2PL)</i>	<ul style="list-style-type: none"> • <i>Two-phase locking (2PL)</i>, • single server: <i>ACID</i>, • distributed systems: <i>BASE</i> 	<i>ACID</i>	<i>ACID</i>	<i>ACID</i> , <i>MVCC</i> – <i>Multiversion Concurrency Control</i>
Replication, skaling	Master-Slave replikation, Auto-Sharding	Master-master replication	Multi-master replication	Multi-Master-Replikation, Sharding	Master-slave replication, sharding
Remarks	<i>BSON</i> with a maximum document size of 16 MB.				

3.9.4 Graph database systems

Graph databases specialise in networked information and the simplest and most efficient possible *Graph traversal*.

Graph model

A graph consists of a number of nodes and edges. Graphs are used to represent a variety of problems through nodes, edges and their relationships, for example in navigation systems in which the paths are stored in the form of graphs.

Graph traversal

Graph traversal is mostly used to find nodes. There are different algorithms for such search queries in a graph, which can be roughly divided into

- Breadth-first search, BFS and depth-first search, DFS

The breadth-first search begins with all neighboring nodes of the start node. In the next step, the neighbors of the neighbors are then searched. The path length increases with each iteration.

The depth-first search follows a path until a node with no outgoing edges is found. The path is then traced back to a node that has further outgoing edges. The search will then continue there.

- Algorithmic traversal

Examples of algorithmic traversal are

- Hamiltonian path (traveling salesman)
- Eulerian path
- Dijkstra's algorithm

- Randomised traversal

The graph is not run through according to a certain scheme, but the next node is selected at random. This allows a search result to be presented much faster, especially with large graphs, but this is not always the best.

Database systems

Typical graph databases are Neo4j, OrientDB and ArangoDB.

Home	Neo4j	OrientDB	ArangoDB
GitHub	neo4j/neo4j	orienttechnologies/orientdb	arangodb/arangodb
Docs	neo4j.com/docs/	orientdb.dev/docs/	arangodb.com/documentation/
Application areas	CMS, social networks, GIS systems, ERP, ...	Master data management, social networks, time series, key value, traffic management	Fraud Detection, IoT, identity management, e-commerce, network, logistics, CMS
Development language	Java	Java	C++, JavaScript
Licenses	AGPL and commercially	Apache License 2.0	Apache License 2.0
Data model	<i>Property graph model</i>	Multi-Model	Multi-model: documents, graphs and <i>Key/value pair</i>
Query language	REST, Cypher, Gremlin	Extended SQL, Gremlin	ArangoDB Query Language (AQL) <i>ACID, MVCC – Multiversion Concurrency Control</i>
Transactions, concurrency	<ul style="list-style-type: none"> <i>Two-phase locking (2PL)</i> single Server: <i>ACID</i> distributed systems: <i>BASE</i> 	<i>ACID</i>	
Replication, scaling	Master-slave with master failover	Multi-master replication, Sharding	Master-slave replication, sharding
Remarks			

See also:

- [Apache TinkerPop Home](#)
- [TinkerPop Documentation](#)
- github.com/apache/tinkerpop
- [Practical Gremlin – An Apache TinkerPop Tutorial](#)
- [gremlinpython](#)

3.9.5 Object database systems

Many programming languages suggest object-oriented programming, so storing these objects seems natural. It therefore makes sense to design the entire process from implementation to storage uniformly and simply. In detail, the advantages are:

Natural modeling and representation of problems

Problems can be modeled in ways that are very close to the human way of thinking.

Clearer, more readable and more understandable

The data and the functions operating on them are combined into one unit, making the programs clearer, more readable and easier to understand.

Modular and reusable

Program parts can be easily and flexibly reused.

Expandable

Programs can be easily expanded and adapted to changed requirements.

Object-relational impedance mismatch

Object-oriented programming and relational data storage are problematic for various reasons. Inheritance is an important concept in OOP for implementing complex models. In the relational paradigm, however, there is nothing like it. Object-relational mappers, ORM, such as *SQLAlchemy*, were developed to convert corresponding class hierarchies into a relational model. In principle there are two different approaches for an ORM, whereby in both cases a table is created for a class:

Vertical partitioning

The table only contains the attributes of the corresponding class and a foreign key for the table of the superclass. An entry is then created for each object in the table belonging to the class and in the tables of all superclasses. When accessing the tables, joins must be used, which can lead to significant performance losses in complex models.

Horizontal partitioning

Each table contains the attributes of the associated class and all superclasses. If the superclass is changed, however, the tables of all derived classes must also be updated.

Basically, when combining OOP and relational data management, two data models must always be created. This makes this architecture significantly more complex, more error-prone and more time-consuming to maintain.

Database systems

Examples of object database systems are ZODB.

Home	ZODB
GitHub	zopefoundation/ZODB
Docs	www.zodb.org/en/latest/tutorial.html
Application areas	Plone, Pyramid, BTrees, volatile data
Development language	Python
Licenses	Zope Public License (ZPL) 2.1
Data model	PersistentList, PersistentMapping, BTree
Query language	
Transactions, concurrency	<i>ACID</i>
Replication, skaling	ZODB Replication Services (ZRS)
Remarks	

3.9.6 XML database systems

XML databases are able to validate XML documents against an XML schema or a DTD. In addition, they support at least *XPATH*, *XQuery* and *XSLT*.

Database systems

Examples of XML database systems are eXist and MonetDB.

Home	eXist	MonetDB	BaseX
GitHub	eXist-db/exist	MonetDB/MonetDB	BaseXdb/basex
Docs	exist-db.org/exist/apps/doc/docume	www.monetdb.org/Documentation	docs.basex.org
Application areas	CMS	CMS, Data-Warehouse, Data mining	CMS
Development language	Java	C	Java
Licenses	LGPL-2.1 License	Mozilla Public License 2.0	BSD-3-Clause License
Data model	XML	XML, column-oriented data structure	XML
Query language	<i>XQuery, XPATH</i>	SQL	<i>XQuery, XPATH</i>
Transactions, concurrency		<i>Optimistic Concurrency</i>	<i>ACID</i> , XQuery Locks
Replication, skaling	Master-slave replication	Transaction replication	
Remarks		With R, analyses can be carried out directly at the database level.	

Major concepts and technologies of NoSQL databases are

- *MapReduce*
- *CAP theorem*
- *Eventual consistency* and *BASE*
- *Consistent hash function*
- *MVCC – Multiversion Concurrency Control*
- *Vector clock*
- *Paxos*

3.10 Application Programming Interface (API)

APIs can be used to provide the data. *FastAPI* is a library that can generate APIs and documentation based on *OpenAPI* and *JSON Schema*. *gRPC*, on the other hand, is a modern open source RPC framework that uses HTTP/2 and QUIC.

To determine the design of your API, you can follow *Zalando’s API Styleguide*. Later, you can use *Zally* to automatically check the quality of your API. You can also define your own rules for Zally, see *Rule Development Manual*.

See also:

- REST API Design – Resource Modeling
- Richardson Maturity Model – steps toward the glory of REST
- Irresistible APIs – Designing web APIs that developers will love
- REST in Practice
- Build APIs You Won’t Hate
- Representational State Transfer (REST)

3.10.1 FastAPI

FastAPI is a web framework for building APIs with Python 3.6+ based type hints.

Key features are:

- very high performance thanks to [pydantic](#) for the data part and [Starlette](#) for the web part.
- fast and easy to code
- validation for most Python data types, including
 - JSON objects (`dict`)
 - JSON array (`list`)
 - string (`str`), defining min and max length
 - numbers (`int`, `float`) with min and max values, etc.
 - URLs
 - email with [python-email-validator](#)
 - UUID
 - ... and others
- robust, production-ready code with automatic interactive documentation
- based on the open standards for APIs: [OpenAPI](#) formerly known as Swagger) and [JSON Schema](#)

See also:

- [Home](#)
- [GitHub](#)

Installation

Requirements

```
$ pipenv install fastapi
Adding fastapi to Pipfile's [packages]...
✓ Installation Succeeded
Locking [dev-packages] dependencies...
✓ Success!
Locking [packages] dependencies...
✓ Success!
...
```

Optional requirements

For production you also need an [ASGI](#) server, such as [uvicorn](#):

```
$ pipenv install uvicorn
Adding uvicorn to Pipfile's [packages]...
✓ Installation Succeeded
Locking [dev-packages] dependencies...
✓ Success!
Locking [packages] dependencies...
✓ Success!
Updated Pipfile.lock (051f02)!
...
```

Pydantic can use the optional dependencies

ujson

for faster JSON parsing.

email_validator

for email validation.

Starlette can use the optional dependencies

httpx

if you want to use the `TestClient`.

aiofiles

if you want to use `FileResponse` or `StaticFiles`.

jinja2

if you want to use the default template configuration.

python-multipart

if you want to support form parsing, with `request.form()`.

itsdangerous

required for `SessionMiddleware` support.

pyyaml

for Starlette's `SchemaGenerator` support.

graphene

for `GraphQLApp` support.

ujson

if you want to use `UJSONResponse`.

orjson

if you want to use `ORJSONResponse`.

They can be installed, e.g. with:

```
$ pipenv install fastapi[ujson]
```

Alternatively you can install all of these with:

```
$ pipenv install fastapi[all]
```

Example

1. Create

Create a file `main.py` with:

```
from typing import Optional

from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root():
    return {"Hello": "World"}

@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}
```

2. Run

Run the server with:

```
$ pipenv run uvicorn main:app --reload
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:      Started reloader process [89155] using statreload
INFO:      Started server process [89164]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

3. Check

Open your browser at <http://127.0.0.1:8000/> and you will see:

JSON			
Rohdaten			
Kopfzeilen			
Speichern	Kopieren	Alle einklappen	Alle ausklappen
🔍 JSON durchsuchen			
Hello: "World"			

You will also get an interactive API documentation provided by [Swagger UI](#) at <http://127.0.0.1:8000/docs>:

You will also get an alternative automatic documentation provided by [ReDoc](#) at <http://127.0.0.1:8000/redoc>:

FastAPI0.1.0OAS3

OpenAPI JSON

default

GET / Read Root

Try it out

Parameters

No parameters

Responses

Code	Description	Links
200	Successful Response	No links

Media type

application/json

Controls Accept header.

Example Value | Schema

(no example available)

GET /items/{item_id} Read Item

Try it out

Parameters

Name	Description
item_id * required integer (path)	item_id
q string (query)	q

Responses

Code	Description	Links
200	Successful Response	No links
422	Validation Error	No links

Media type

application/json

Controls Accept header.

Example Value | Schema

(no example available)

Media type

application/json

Controls Accept header.

Example Value | Schema

```
{
  "detail": {
    {
      "loc": [
        "string"
      ],
      "msg": "string",
      "type": "string"
    }
  }
}
```

Schemas

HTTPValidationError {

detail {

Detail {

title: Detail

ValidationError > {...}

}

}

ValidationError {

loc {

msg {

type {

Location > [...]

string

title: Message

string

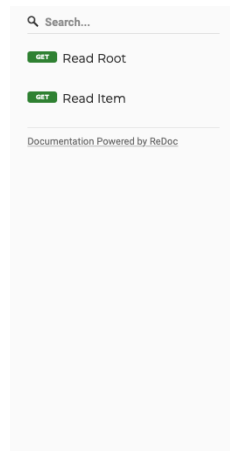
title: Error type

}

}

}

}



FastAPI (0.1.0)

Download OpenAPI specification: [Download](#)

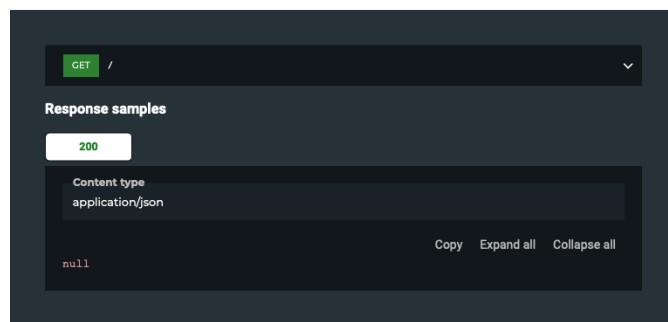
Read Root

Responses

✓ 200 Successful Response

RESPONSE SCHEMA: `application/json`

any



Read Item

PATH PARAMETERS

→ `item_id` required integer (Item Id)

QUERY PARAMETERS

→ `q` string (Q)

Responses

✓ 200 Successful Response

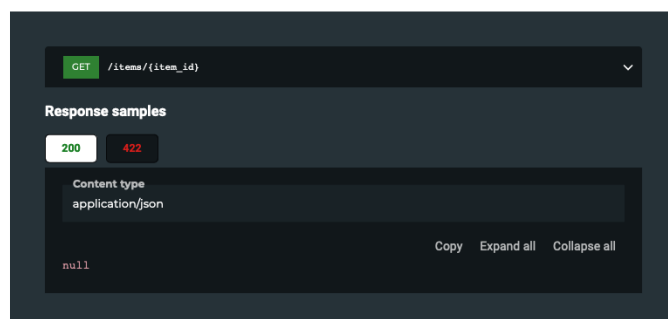
RESPONSE SCHEMA: `application/json`

any

✓ 422 Validation Error

RESPONSE SCHEMA: `application/json`

→ `detail` > Array of objects (Detail)



4. Update

Now we modify the file `main.py` to receive a body from a PUT request:

```
from typing import Optional

from pydantic import BaseModel

from fastapi import FastAPI

app = FastAPI()

class Item(BaseModel):
    name: str
    price: float
    is_offer: Optional[bool] = None

@app.get("/")
def read_root():
    return {"Hello": "World"}

@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}

@app.put("/items/{item_id}")
def update_item(item_id: int, item: Item):
    return {"item_name": item.name, "item_id": item_id}
```

The server should reload the file automatically because we added `--reload` to the `uvicorn` command. Also the interactive API documentation will show the new body with PUT. If you click on the button *Try it out* you will fill in the parameter for `item_id`. Then click on the *Execute* button and the your browser will send the parameter to the API and show them on the screen, for example as response body:

```
{
  "item_name": "string",
  "item_id": 1234
}
```

Tips

Project structure

If you want to use

- [SQLModel](#) for Python database interaction (ORM)
- [Pydantic](#) for data validation
- [PostgreSQL](#) for data storage

the data structure could look like this:

```
fastapi-example
├── LICENSE
├── README.rst
├── alembic.ini
├── app
│   ├── __init__.py
│   ├── alembic
│   │   ├── README
│   │   ├── env.py
│   │   ├── script.py.mako
│   │   └── versions
│   │       └── 3512b954651e_initialize_models.py
│   ├── api
│   │   ├── __init__.py
│   │   ├── deps.py
│   │   ├── main.py
│   │   └── routes
│   │       ├── __init__.py
│   │       ├── items.py
│   │       └── utils.py
│   ├── core
│   │   ├── __init__.py
│   │   ├── config.py
│   │   └── db.py
│   ├── crud.py
│   ├── main.py
│   ├── models.py
│   └── tests
│       ├── __init__.py
│       ├── api
│       │   ├── __init__.py
│       │   └── routes
│       │       ├── __init__.py
│       │       └── test_items.py
│       ├── conftest.py
│       └── crud
│           ├── __init__.py
│           └── test_items.py
└── pyproject.toml
```

Extensions

Administration

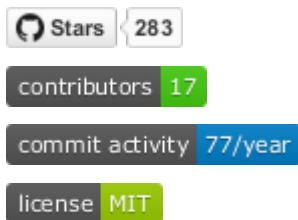
SQLAlchemy Admin for Starlette/FastAPI

Flexible admin interface for *SQLAlchemy* models.



Piccolo Admin

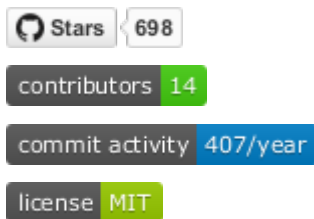
Simple but powerful admin interface over Piccolo tables that lets you easily add, edit and filter your data



Authentication

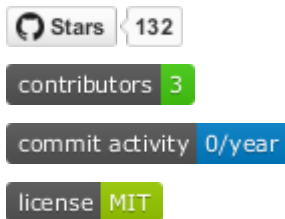
AuthX

Ready-to-use and customisable authentication and OAuth2 management



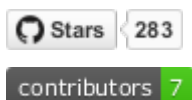
FastAPI Security

Authentication and authorisation



FastAPI simple security

API key-based security package focused on ease of use



commit activity 22/year
license MIT

FastAPI Users

Quickly adds a customisable registration and authentication system

Stars 4.1k
contributors 61
commit activity 55/year
license MIT

ORMs

FastAPI-SQLAlchemy

Easy integration between FastAPI, *SQLAlchemy* and application

Stars 581
contributors 7
commit activity 43/year
license MIT

FastAPIwee

Easy way to create a REST API based on *PeeWee* models

Stars 17
contributors 1
commit activity 0/year
license MIT

GINO

Lightweight asynchronous ORM built on SQLAlchemy Core for Python *asyncio*, supporting PostgreSQL with *asyncpg*, and MySQL with *aiomysql* (→ [example](#))

Stars 2.6k
contributors 36
commit activity 0/year
license not identifiable by github

ORM

async ORM, which builds on SQLAlchemy Core, *Databases* and *TypeSystem*

Stars 1.8k
contributors 16

commit activity 77/year
 license MIT

ormar

Asynchronous mini-ORM, with which you only need to maintain one set of models and migrate them with *Alembic* if necessary (→ [example](#)); it is also supported by *fastapi-users*, *fastapi-crudrouter* and *fastapi-pagination*

Stars 1.6k
 contributors 31
 commit activity 128/year
 license MIT

Piccolo

Fast, user-friendly ORM and query builder that supports Asyncio (→ [examples](#))

Stars 1.3k
 contributors 39
 commit activity 91/year
 license MIT

Prisma Client Python

Building on the TypeScript ORM *Prisma* with support for PostgreSQL, MySQL, SQLite, MongoDB and SQL Server (→ [Example](#))

Stars 1.3k
 contributors 17
 commit activity 77/year
 license MIT

Tortoise ORM

Easy-to-use asyncio ORM inspired by Django (→ [examples](#)); *Aerich* is a database migration tool for Tortoise ORM.

Stars 4.3k
 contributors 122
 commit activity 43/year
 license Apache-2.0

SQLModel

Library for the interaction of SQL databases with Python objects

Stars 13k
 contributors 72

commit activity 168/year
license MIT

SQL Query Builders

FastAPI Filter

Querystring filters for the Api endpoints and the Swagger user interface. The supported backends are *SQLAlchemy* and *MongoEngine*.

Stars 178
contributors 10
commit activity 145/year
license MIT

asyncpgsa

Python wrapper around *asyncpg* for use with *SQLAlchemy*

Stars 412
contributors 30
commit activity 0/year
license Apache-2.0

Databases

Simple asyncio support for the database drivers *asyncpg*, *aiopg*, *aiomysql*, *asyncmy* and *aiosqlite*

Stars 3.7k
contributors 55
commit activity 16/year
license BSD-3-Clause

ODMs

Beanie

Asynchronous Python object document mapper (ODM) for MongoDB, based on *Motor* and *Pydantic*

Stars 1.8k
contributors 74
commit activity 85/year
license Apache-2.0

MongoEngine

Python Object-Document Mapper for working with MongoDB

 Stars 4.2k

contributors 290

commit activity 71/year

license MIT

ODMantic

Asynchronous ODM (Object Document Mapper) for MongoDB based on Python type hints and [pydantic](#)

 Stars 978

contributors 19

commit activity 81/year

license ISC

Code generators

fastapi-code-generator

Code generator creates a FastAPI application from an openapi file, using [datamodel-code-generator](#) to generate the pydantic model

 Stars 914

contributors 23

commit activity 91/year

license MIT

FastAPI-based API Client Generator

mypy- and IDE-friendly API client from an OpenAPI specification using the [OpenAPI Generator](#)

 Stars 323

contributors 6

commit activity 0/year

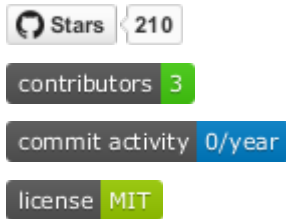
license Apache-2.0

Utilities

Caching

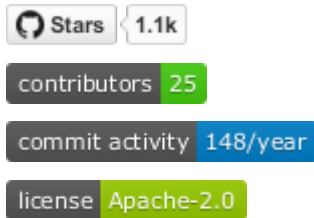
FastAPI Cache

Lightweight cache system



fastapi-cache

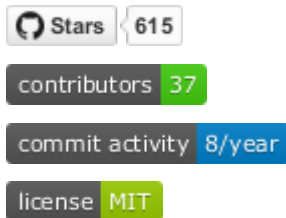
Caching of fastapi responses and function results, with backends supporting *redis*, *memcache* and *dynamodb*



E-mail

Fastapi-mail

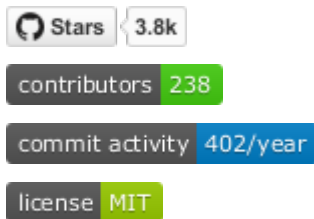
Easy mail system for sending e-mails and attachments, individually or in large quantities



GraphQL

Strawberry GraphQL

Python GraphQL library based on data classes



Logging

ASGI Correlation ID middleware

Middleware to load or generate correlation IDs for each incoming request

 Stars 345

contributors 12

commit activity 21/year

license MIT

starlette context

Middleware for Starlette that allows you to store and access the contextual data of a request

 Stars 419

contributors 6

commit activity 14/year

license MIT

Prometheus

Prometheus FastAPI Instrumentator

Configurable and modular Prometheus instrumentator

 Stars 817

contributors 24

commit activity 24/year

license ISC

starlette_exporter

Prometheus export programme for Starlette and FastAPI

 Stars 292

contributors 18

commit activity 46/year

license Apache-2.0

Starlette Prometheus

Prometheus integration for Starlette

 Stars 266

contributors 16

commit activity 6/year

license **GPL-3.0**

Templating

fastapi-jinja

Integration of the Jinja template language

 Stars **64**

contributors **5**

commit activity **0/year**

license **MIT**

fastapi-chameleon

Integration of the template language Chameleon

 Stars **137**

contributors **6**

commit activity **1/year**

license **MIT**

Pagination

FastAPI Pagination

Easy-to-use pagination for FastAPI with integration in sqlalchemy, gino, databases and ormar, among others

 Stars **1k**

contributors **42**

commit activity **793/year**

license **MIT**

Websockets

fastapi-socketio

Easy integration of socket.io in into your FastAPI application

 Stars **322**

contributors **7**

commit activity **0/year**

license **Apache-2.0**

FastAPI Websocket Pub/Sub

Fast and permanent pub/sub channel via websockets

 Stars 452

contributors 9

commit activity 23/year

license MIT

FASTAPI Websocket RPC

Fast and permanent bidirectional JSON RPC channel via websockets

 Stars 182

contributors 10

commit activity 48/year

license MIT

Other tools

Pydantic-SQLAlchemy

Creating Pydantic models from SQLAlchemy models

 Stars 1.1k

contributors 4

commit activity 8/year

license MIT

Fastapi Camelcase

Provision of a class of request and response bodies for FastAPI

 Stars 68

contributors 3

commit activity 8/year

license MIT

fastapi_profiler

FastAPI middleware based on [pyinstrument](#) for performance testing

 Stars 196

contributors 3

commit activity 3/year

license MIT

fastapi-versioning

API versioning for FastAPI web applications

 Stars 617

contributors 14

commit activity 0/year

license MIT

Jupyter Notebook REST API

Run Jupyter notebooks as REST API endpoint

 Stars 75

contributors 2

commit activity 0/year

license MIT

manage-fastapi

Project generator and manager for FastAPI

 Stars 1.6k

contributors 7

commit activity 5/year

license MIT

msgpack-asgi

Automatic negotiation of MessagePack content in ASGI applications

 Stars 283

contributors 3

commit activity 0/year

license MIT

fastapi-plugins

Production-ready plug-ins for the FastAPI framework, including for caching with memcached or Redis, scheduler, configuration and logging

 Stars 339

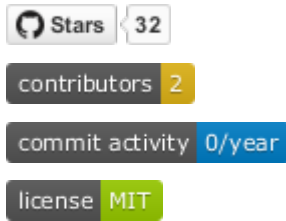
contributors 3

commit activity 6/year

license MIT

fastapi-serviceutils

Optimised logging, exception handling and configuration



3.10.2 gRPC

gRPC is a modern, open source, high-performance remote procedure call (RPC) framework. By default, gRPC uses *Protocol Buffers (Protobuf)* as the Interface Definition Language (IDL) for describing both the service interface and the structure of the payload messages. In gRPC, a client application can directly call a method on a server application on a different machine as if it were a local object, making it easier for you to create distributed applications and services. As in many RPC systems, gRPC is based on the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types. The server implements the interface and runs a gRPC server to handle client calls; the client has a stub that provides the same methods as the server.

The following are the main design principles of gRPC:

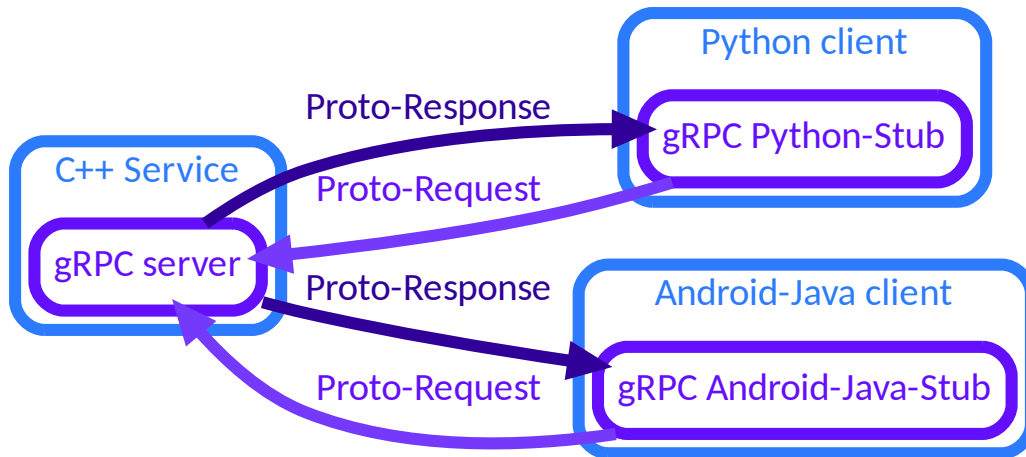
- gRPC can be created on all common development platforms and in many different languages.
- It is designed to work on devices with low CPU and memory capabilities, such as Android¹ and iOS devices, MicroPython boards and browsers²³.
- It is licensed under Apache License 2.0 and uses open standards such as HTTP/2 and Quick UDP Internet Connections (QUIC).
- gRPC is interoperable and can therefore also be used in the LoRaWan (Long Range Wide Area Network), for example.
- The individual layers can be developed independently of each other. For example, the transport layer (OSI layer 4) can be developed independently of the application layer (OSI layer 7).
- gRPC supports various serialisation formats, including *Protocol Buffers (Protobuf)*, *JSON*⁴, *XML/HTML* and *Thrift*
- Asynchronous and synchronous (blocking) processing are supported in most languages.
- Streaming of messages in a single RPC call is supported.
- gRPC allows protocol extensions for security, health checks, load balancing, failover, etc.

¹ gRPC in Android Java

² gRPC-Web is Generally Available

³ gRPC-Web Client Runtime Library

⁴ gRPC + JSON



Starting with an interface definition in a `.proto` file, gRPC provides Protocol Compiler plugins that generate Client- and Server-side APIs. Both synchronous and asynchronous communication is supported in most languages. gRPC also supports streaming of messages in a single RPC call. The [gRPC protocol](#) abstractly specifies the communication between clients and servers:

1. First the stream is started by the client with a mandatory `Call Header`
 1. followed by optional `Initial-Metadata`
 2. followed by optional `Payload Messages`.

The contents of `Call Header` and `Initial Metadata` are sent as HTTP/2 headers compressed with HPACK.

2. The server answers with an optional `Initial-Metadata`
 1. followed by `Payload Messages`
 2. and terminated with mandatory `Status` and optional `Status-Metadata`.

`Payload Messages` are serialised into a byte stream fragmented into HTTP/2 frames. `Status` and `Trailing-Metadata` are sent as HTTP/2 trailing headers.

Unlike [FastAPI](#), however, the gRPC API cannot simply be tested on the command line with `cURL`. If necessary, you can use [grpcurl](#). This requires that the gRPC server supports the [gRPC Server Reflection Protocol](#). Usually *Reflection* should only be available in the development phase. Then you can call `grpcurl`, e.g. with:

```
$ grpcurl localhost:9111 list
```

See also:

- [Home](#)
- [GitHub](#)
- [gRPC Blog](#)

gRPC-Example

By default, gRPC uses *Protocol Buffers (Protobuf)* for serialising data, although it also works with other data formats such as JSON.

Define the data structure

The first step when working with protocol buffers is to define the structure for the data you want to serialise in a `.proto` file. Protocol buffer data is structured as *messages*, where each message is a small logical record of information containing a series of name-value pairs called *fields*. Here's a simple example `accounts.proto`:

Listing 2: `accounts.proto`

```
// SPDX-FileCopyrightText: 2021 Veit Schiele
//
// SPDX-License-Identifier: BSD-3-Clause

syntax = "proto3";

message Account {
```

Warning: You shouldn't simply use `uint32` for user or group IDs, as these would be far too easy to guess. You can use an [RFC 4122](#)-compliant implementation for this purpose. You can find a corresponding protobuf configuration in `rfc4122.proto`.

After you have defined your data structure, you use the protocol buffer compiler `protoc` to generate descriptors in your preferred languages. These provide simple accessors for each field, as well as methods to serialise the whole structure. For example, if your language is Python, running the compiler on the example above will generate declarators you can then use in your application to populate, serialise, and retrieve protocol buffer messages.

Define the gRPC service

gRPC services are also defined in the `.proto` files, with RPC method parameters and return types specified as protocol buffer messages:

Listing 3: `accounts.proto`

```
uint32 account_id = 1;
string account_name = 2;
}

message CreateAccountRequest {
    string account_name = 1;
}

message CreateAccountResult {
    Account account = 1;
}

message GetAccountsRequest {
```

(continues on next page)

(continued from previous page)

```

    repeated Account account = 1;
}

message GetAccountsResult {
    Account account = 1;
}

service Accounts {
    rpc CreateAccount (CreateAccountRequest) returns (CreateAccountResult);
    rpc GetAccounts (GetAccountsRequest) returns (stream GetAccountsResult);
}

```

Generate the gRPC Code

```

$ pipenv install grpcio grpcio-tools
$ pipenv run python -m grpc_tools.protoc --python_out=. --grpc_python_out=. accounts.
  ↪ proto

```

This generates two files:

accounts_pb2.py

contains classes for the messages defined in `accounts.proto`.

accounts_pb2_grpc.py

contains the defined classes `AccountsStub` for calling RPCs, `AccountsServicer` for the API definition of the service and a function `add_AccountsServicer_to_server` for the server.

Create server

For this we write the file `accounts_server.py`:

Listing 4: `accounts_server.py`

```

# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

import logging
from concurrent import futures

import accounts_pb2 as accounts_messages
import accounts_pb2_grpc as accounts_service
import grpc

class AccountsService(accounts_service.AccountsServicer):
    def CreateAccount(self, request, context):
        metadata = dict(context.invocation_metadata())
        print(metadata)
        account = accounts_messages.Account(
            account_name=request.account_name, account_id=1

```

(continues on next page)

(continued from previous page)

```

    )
    return accounts_messages.CreateAccountResult(account=account)

def GetAccounts(self, request, context):
    for account in request.account:
        account = accounts_messages.Account(
            account_name=account.account_name,
            account_id=account.account_id,
        )
        yield accounts_messages.GetAccountsResult(account=account)

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    accounts_service.add_AccountsServicer_to_server(AccountsService(), server)
    server.add_insecure_port("[::]:8081")
    server.start()
    server.wait_for_termination()

if __name__ == "__main__":
    logging.basicConfig()
    serve()

```

Create client

For this we create `accounts_client.py`:

Listing 5: `accounts_client.py`

```

# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

import logging

import accounts_pb2 as accounts_messages
import accounts_pb2_grpc as accounts_service
import grpc

def run():
    channel = grpc.insecure_channel("localhost:8081")
    stub = accounts_service.AccountsStub(channel)
    response = stub.CreateAccount(
        accounts_messages.CreateAccountRequest(account_name="tom"),
    )
    print("Account created:", response.account.account_name)

if __name__ == "__main__":

```

(continues on next page)

(continued from previous page)

```
logging.basicConfig()
run()
```

Run client and server

1. Starting the server:

```
$ pipenv run python accounts_server.py
```

2. Starting the client from another terminal:

```
$ pipenv run python accounts_client.py
Account created: tom
```

Test gRPC

pytest-grpc

gRPC can be tested automatically with `pytest-grpc`.

1. First, we install

```
$ pipenv install pytest-grpc
Installing pytest-grpc...
Adding pytest-grpc to Pipfile's [packages]...
✓ Installation Succeeded
...
```

2. Then we create a `Test Fixture` for our *gRPC-Example* with:

Listing 6: tests/test_accounts.py

```
# SPDX-License-Identifier: BSD-3-Clause

from pathlib import Path

import grpc
import pytest
from accounts_pb2 import CreateAccountRequest, GetAccountsRequest

@pytest.fixture(scope="module")
def grpc_add_to_server():
    from accounts_pb2_grpc import add_AccountsServicer_to_server

    return add_AccountsServicer_to_server

@pytest.fixture(scope="module")
def grpc_servicer():
```

(continues on next page)

(continued from previous page)

```

    from accounts_server import AccountsService

    return AccountsService()

@pytest.fixture(scope="module")

```

See also:

- [pytest fixtures](#)

3. Afterwards we can write tests, for example:

```

    return AccountsStub(grpc_channel)

def test_create_account(grpc_stub):
    value = "test-data"
    nl = "\n"

```

4. Authentication can also be tested, for example with:

```

# SPDX-FileCopyrightText: 2021 Veit Schiele
#

assert response.name == f"test-{request.name}"

@pytest.fixture(scope="module")
def grpc_server(_grpc_server, grpc_addr, my_ssl_key_path, my_ssl_cert_path):
    """
    Overwrites default `grpc_server` fixture with ssl credentials
    """
    credentials = grpc.ssl_server_credentials(
        [(my_ssl_key_path.read_bytes(), my_ssl_cert_path.read_bytes())]
    )

    _grpc_server.add_secure_port(grpc_addr, server_credentials=credentials)
    _grpc_server.start()
    yield _grpc_server
    _grpc_server.stop(grace=None)

@pytest.fixture(scope="module")
def my_channel_ssl_credentials(my_ssl_cert_path):
    # If we're using self-signed certificate it's necessarily to pass root_
    ↪ certificate to channel
    return grpc.ssl_channel_credentials(root_certificates=my_ssl_cert_path.read_
    ↪ bytes())

@pytest.fixture(scope="module")

```

(continues on next page)

(continued from previous page)

```

def grpc_channel(my_channel_ssl_credentials, create_channel):
    """
    Overwrites default `grpc_channel` fixture with ssl credentials
    """
    with create_channel(my_channel_ssl_credentials) as channel:
        yield channel

@pytest.fixture(scope="module")
def grpc_authored_channel(my_channel_ssl_credentials, create_channel):
    """
    Channel with authorization header passed
    """
    grpc_channel_credentials = grpc.access_token_call_credentials("some_token")
    composite_credentials = grpc.composite_channel_credentials(
        my_channel_ssl_credentials, grpc_channel_credentials
    )
    with create_channel(composite_credentials) as channel:
        yield channel

@pytest.fixture(scope="module")
def my_authored_stub(grpc_stub_cls, grpc_channel):
    """
    Stub with authorized channel
    """
    return grpc_stub_cls(grpc_channel)

```

5. Afterwards we can test against a real gRPC server with:

```
$ pipenv run pytest --fixtures tests/
```

or directly against the Python code:

```

$ pipenv run pytest --fixtures tests/ --grpc-fake-server
===== test session starts =====
platform darwin -- Python 3.7.3, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: /Users/veit/cusy/trn/Python4DataScience/docs/data/grpc
plugins: grpc-0.8.0
collected 2 items

tests/test_accounts.py .F                                     [100%]
...

```

See also:

- [GitHub](#)
- [Example](#)

Wireshark

Wireshark is an open source tool for analysing network protocols. In the following, we will show you how to use the gRPC and Protobuf dissectors. They make it easier for you to decode gRPC messages that are serialised in *Protobuf* or *JSON* format. You can also use them to analyse server, client and bidirectional gRPC streaming.

Note: Usually, Wireshark can only analyse gRPC messages in plain text. For dissecting a TLS session, Wireshark needs the secret key, the export of which is currently only supported by [Go gRPC](#)¹.

See also:

- [Analyzing gRPC messages using Wireshark](#)
-

3.11 Glossary

ACID

ACID is an acronym for **A**tomicity **C**onsistency **I**solation **D**urability. They are a prerequisite for the reliability of database transactions.

Atomicity

A transaction is a series of database operations that are either carried out completely or not at all.

Consistency

Transaction that leaves a consistent state after completion. The integrity conditions defined in the database schema are checked before the transaction is completed.

Isolation

Concurrent transactions must not influence each other. This is usually achieved with *Locking*, which restricts the concurrency.

Durability

After a successful transaction, data must be permanently stored in the database and can be secured, for example, by writing a transaction log.

BASE

BASE is an acronym for **B**asically Available, **S**oft State, **E**ventually Consistent and originated as the opposite of *ACID*.

A very optimistic concept of consistency is used that does not require *Locking*. Locks are problematic in several ways, since access is not possible as long as data records are locked by other transactions. In addition, the agreement to set a lock is already very complex.

Data consistency is seen as a state that can be achieved at some point. This is the idea of *Eventual Consistency*.

With BASE, competing access is avoided through *MVCC – Multiversion Concurrency Control*. However, there is a wide range of solutions for the various distributed database systems:

- Causal Consistency
is comparable to the consistency in *ACID*.
- Read Your Writes
- Session Consistency

¹ [How to Export TLS Master keys of gRPC](#)

- Monotonic Read Consistency
- Monotonic Write Consistency

CAP theorem

CAP is an acronym for **C**onsistency, **A**vailability and **P**artition Tolerance. The findings of the CAP theorem play a central role in the selection of a distributed database system.

The CAP theorem states that in distributed systems the three requirements of consistency, availability and failure tolerance are not fully compatible and only a maximum of two out of three can be achieved. Therefore it must be decided individually for each application whether a CA, CP or AP application should be implemented.

Cassandra

Cassandra is a *Column-oriented database systems*, and was originally developed by Facebook to optimise searches in email. Today it is further developed under the umbrella of the [Apache Software Foundation](#).

Cassandra's data model has neither a logical structure nor a schema. For the modeling it is recommended «*First write your queries then model your data*». Then usually a *Column Family* is created for each expected request. The data is denormalised, but each column family responds to a specific type of query.

In Cassandra, the consistency can be specified for each request. This allows specific requests to be very consistent while others sacrifice consistency for speed. There are, for example, the following four levels for write consistency:

ANY

ensures that the data is stored in at least one node.

ONE

ensures that the data is stored in the commit log of at least one replica.

QUORUM

ensures that the data is stored in a quorum of replicas.

ALL

ensures that the data is saved on all replicas.

Cassandra provides two different APIs: [Thrift](#) and [CQL \(Cassandra Query Language\)](#).

Column Family

Column families correspond to tables in relational databases. They group columns with the same or similar content, for example

```
profile = {
    cusy: {
        name:      "cusy GmbH",
        email:     "info@cusy.io",
        website:   "cusy.io"
    },
    veit: {
        name:      "Veit Schiele",
        email:     "veit.schiele@cusy.io",
    }
}
```

Consistent hash function

Consistent hash functions minimise the number of reallocations, since not all keys have to be reallocated when a change occurs, only the size of a hash table is changed.

Consistency

The state of a database is said to be consistent if the stored data meets all requirements for *Semantic integrity*.

CouchDB

CouchDB an acronym for Cluster of **un**reliable commodity hardware **D**ata **B**ase. This is a *Document-oriented database systems*.

Eventual Consistency

«Consistency as a state transition that is reached at some point.»

The term was developed for *BASE* as an alternative to *ACID*.

Graph traversal

Graph traversal is mostly used to find nodes. There are different algorithms for such search queries in a graph, which can be roughly divided into

- Breadth-first search, BFS and depth-first search, DFS

The breadth-first search begins with all neighboring nodes of the start node. In the next step, the neighbors of the neighbors are then searched. The path length increases with each iteration.

The depth-first search follows a path until a node with no outgoing edges is found. The path is then traced back to a node that has further outgoing edges. The search will then continue there.

- Algorithmic traversal

Examples of algorithmic traversal are

- Hamiltonian path (traveling salesman)
- Eulerian path
- Dijkstra's algorithm

- Randomised traversal

The graph is not run through according to a certain scheme, but the next node is selected at random. This allows a search result to be presented much faster, especially with large graphs, but this is not always the best.

Graph model

A graph consists of a number of nodes and edges. Graphs are used to represent a variety of problems through nodes, edges and their relationships, for example in navigation systems in which the paths are stored in the form of graphs.

Graph partitioning

With graph partitioning, graphs are divided into smaller subgraphs. However, there is no mathematically exact method to minimise the number of intersected edges, but only a few heuristic algorithms, for example clustering algorithms, which combine strongly networked subgraphs to abstract nodes.

One speaks of overlapping partitioning in the case of graphs that cannot be completely divided and exist in several subgraphs.

HBase

HBase is a *Column-oriented database systems*, which is based on distributed file systems and is designed for real-time access to large databases.

Hypertable

Hypertable is a *Column-oriented database systems* and is based on distributed file systems. The data model is that of a multi-dimensional table that can be searched using keys. The first dimension is the so-called *row key*, the second is the *Column family*, the third dimension is the *column qualifier* and the fourth dimension is time.

Key/value pair

A value is always assigned to a specific key, which can consist of a structured or arbitrary character string. These keys can be divided into namespaces and databases. In addition to strings, the values can also contain lists, sets or hashes.

Locking

Locking is the term used to describe the blocking of data for concurrent transactions.

There are different lock procedures, depending on the type of access:

- *Optimistic concurrency*
- *Pessimistic locking*
- *Two-phase locking (2PL)*

MapReduce

MapReduce is a framework introduced by Google Inc. in 2004, which is used for the concurrent computations of enormous amounts of data on computer clusters. It was inspired by the *map* and *reduce* functions, which are often used in functional programming, even if the semantics deviate slightly from them.

MongoDB

MongoDB is a schema-free *Document-oriented database systems*, that manages documents in *BSON* format.

MVCC – Multiversion Concurrency Control

MVCC controls concurrent accesses to data records (read, insert, change, delete) by different, unchangeable versions of these data records. The various versions are arranged in a chronological order, with each version referring to its previous version. MVCC has developed into a central basic technology for NoSQL databases in particular, which makes it possible to coordinate competing accesses even without locking data records.

Optimistic concurrency

Optimistic concurrency, also called optimistic locking, is a form of *locking*, which assumes that there are few write accesses to the database and read accesses do not trigger a lock. In the event of changes, a check is first made to determine whether the time stamp has remained unchanged since the data was read.

Paxos

Paxos is a family of protocols for building consensus on a network of unreliable or fallible processors.

Pessimistic locking

Pessimistic *locking* assumes a lot of write accesses to the database. Read access is therefore also blocked. The data is only released again when the changes have been saved.

Property graph model

PGM

Nodes and edges consist of objects with properties embedded in them. Not only a value (label) is stored in an edge or a node, but a *Key/value pair*.

Riak

In essence, Riak is a decentralised *Key/value pair* with a flexible *MapReduce* engine.

Redis

Redis is a *Key-value database systems*, that usually stores all data in RAM.

Semantic integrity

Semantic integrity is always given when the entries are correct and consistent. Then we talk of consistent data. If this is not the case, the data is inconsistent. In SQL, the semantic integrity can be checked with TRIGGER and CONSTRAINT

Two-phase locking (2PL)

The two-phase *locking* protocol distinguishes between two phases of transactions:

1. The growth phase in which locks can only be set but not released.
2. The shrinkage phase, in which locks can only be released but not requested.

The two-phase lock protocol knows three lock states:

SLOCK, shared lock or read lock

is set with read access to data

XLOCK, exclusive lock or write lock

is set with write access to data

UNLOCK

removes the locks SLOCK and XLOCK.

Vector clock

A vector clock is a software component used to assign unique time stamps to messages. It allows a causal order to be assigned to the events in distributed systems on the basis of a time stamp and, in particular, to determine the concurrency of events.

XPATH

XPATH processes the tree structure of an XML document and generates extracts from XML documents. In order to receive complete XML documents as a result, these must be created with *XQuery* or *XSLT*, for example. XPATH is not a complete query language as it is limited to selections and extractions.

XPATH has been part of *XQuery* since version 1.1 and from version 2.0 onwards, XPATH is extended by *XQuery*.

XQuery

XQuery stands for *XML Query Language* and is mainly a functional language in which nested expressions can also be evaluated during a query.

XSLT



















XSLT is an acronym for **Extensible Stylesheet Language Transformation**. It can be used to transform XML documents.

DATA CLEANSING AND VALIDATION

In the following, we want to give you a practical overview of various libraries and methods for [data cleansing](#) and validation with Python. Besides well-known libraries like NumPy and Pandas, we also use several small, specialised libraries like [dedupe](#), [fuzzywuzzy](#), [voluptuous](#), [bulwark](#), [tdda](#) and [hypothesis](#). We prefer these more lightweight solutions to large, universal systems like [Great Expectations](#) or [MobyDQ](#).

4.1 Overview

Table 1: GitHub-Insights

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
fuzzywuzzy	 Stars 9.1k	contributors 60	commit activity 0/year	license GPL-2.0
dedupe	 Stars 4k	contributors 50	commit activity 30/year	license MIT
Bulwark	 Stars 221	contributors 7	commit activity 0/year	license LGPL-3.0
Hypothesis	 Stars 7.3k	contributors 285	commit activity 1.4k/year	license not identifiable by github
TDDA	 Stars 276	contributors 8	commit activity 25/year	license MIT
Voluptuous	 Stars 1.8k	contributors 73	commit activity 21/year	license BSD-3-Clause
scikit-learn	 Stars 58k	contributors 408	commit activity 1.3k/year	license BSD-3-Clause
pandera	 Stars 3k	contributors 121	commit activity 171/year	license MIT
Validr	 Stars 211	contributors 4	commit activity 31/year	license not identifiable by github
marshmallow	 Stars 6.9k	contributors 177	commit activity 149/year	license MIT
datacleaner	 Stars 1k	contributors 3	commit activity 0/year	license MIT
Probatus	 Stars 122	contributors 25	commit activity 30/year	license MIT
popmon	 Stars 487	contributors 11	commit activity 50/year	license MIT
Pandas Profiling	 Stars 12k	contributors 102	commit activity 155/year	license MIT
pandas-validation	 Stars 20	contributors 1	commit activity 0/year	license MIT
PandasSchema	 Stars 185	contributors 6	commit activity 0/year	license GPL-3.0
Opulent-Pandas	 Stars 9	contributors 1	commit activity 0/year	license Apache-2.0
signpost	 Stars 1	contributors 1	commit activity 0/year	license GPL-3.0

4.1.1 Managing missing data with pandas

Missing data often occurs in data analyses. pandas simplifies working with missing data as much as possible. For example, all *descriptive statistics* of pandas objects exclude missing data by default. pandas uses the floating point value NaN (*Not a Number*) for numerical data to represent missing data.

Methods for handling NA objects:

Argument	Description
dropna	filters axis labels based on whether values for each label have missing data, with different thresholds for the amount of missing data to tolerate
fillna	fills missing data with a value or with an interpolation method such as <code>ffill</code> or <code>bfill</code>
isna	returns boolean values indicating which values are missing/NA
notna	negates <code>isna</code> and returns <code>True</code> for non-NA values and <code>False</code> for NA values

This notebook introduces some ways to manage missing data using Pandas DataFrames. For more information, see the Pandas documentation: [Working with missing data](#) and [Missing data cookbook](#).

See also

- [Dora](#)
- [Badfish](#)

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

```
[2]: df = pd.read_csv(
    "https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/iot_example_
    ↪with_nulls.csv"
)
```

1. Check the data

In pandas, a convention borrowed from the R programming language was adopted and missing data was referred to as NA, which stands for *not available*. In statistical applications, NA data can be either data that does not exist or data that exists but has not been observed (for example due to problems in data collection). When cleaning data for analysis, it is often important to analyse the missing data itself to identify problems in data collection or potential biases in the data due to the missing data. First we display the first 20 data records:

```
[3]: df.head(20)
```

```
[3]:
```

	timestamp	username	temperature	heartrate	\
0	2017-01-01T12:00:23	michaelsmith	12.0	67	
1	2017-01-01T12:01:09	kharrison	6.0	78	
2	2017-01-01T12:01:34	smithadam	5.0	89	
3	2017-01-01T12:02:09	eddierodriguez	28.0	76	
4	2017-01-01T12:02:36	kenneth94	29.0	62	
5	2017-01-01T12:03:04	bryanttodd	13.0	86	
6	2017-01-01T12:03:51	andrea98	17.0	81	
7	2017-01-01T12:04:35	scott28	16.0	76	
8	2017-01-01T12:05:05	hillpamela	5.0	82	
9	2017-01-01T12:05:41	moorejefrey	25.0	63	
10	2017-01-01T12:06:21	njohnson	NaN	63	
11	2017-01-01T12:06:53	gsutton	29.0	80	
12	2017-01-01T12:07:41	jessica48	22.0	83	
13	2017-01-01T12:08:08	hornjohn	16.0	73	
14	2017-01-01T12:08:35	gramirez	24.0	73	
15	2017-01-01T12:09:05	schmidtsamuel	NaN	78	
16	2017-01-01T12:09:48	derrick47	NaN	63	

(continues on next page)

(continued from previous page)

17	2017-01-01T12:10:23	beckercharles	12.0	61
18	2017-01-01T12:10:57	ipittman	11.0	69
19	2017-01-01T12:11:34	sabrina65	22.0	82

	build	latest	note
0	4e6a7805-8faa-2768-6ef6-eb3198b483ac	0.0	interval
1	7256b7b0-e502-f576-62ec-ed73533c9c84	0.0	wake
2	9226c94b-bb4b-a6c8-8e02-cb42b53e9c90	0.0	NaN
3	NaN	0.0	update
4	122f1c6a-403c-2221-6ed1-b5caa08f11e0	NaN	NaN
5	0897dbe5-9c5b-71ca-73a1-7586959ca198	0.0	interval
6	1c07ab9b-5f66-137d-a74f-921a41001f4e	1.0	NaN
7	7a60219f-6621-e548-180e-ca69624f9824	NaN	interval
8	a8b87754-a162-da28-2527-4bce4b3d4191	1.0	NaN
9	585f1a3c-0679-0ffe-9132-508933c70343	0.0	wake
10	e09b6001-125d-51cf-9c3f-9cb686c19d02	NaN	NaN
11	607c9f6e-2bdf-a606-6d16-3004c6958436	1.0	update
12	03e1a07b-3e14-412c-3a69-6b45bc79f81c	NaN	update
13	NaN	0.0	interval
14	NaN	0.0	wake
15	b9890c1e-79d5-8979-63ae-6c08a4cd476a	0.0	NaN
16	b60bd7de-4057-8a85-f806-e6eec1350338	NaN	interval
17	b1dacc73-c8b7-1d7d-ee02-578da781a71e	0.0	test
18	1aef7db8-9a3e-7dc9-d7a5-781ec0efd200	NaN	user
19	8075d058-7dae-e2ec-d47e-58ec6d26899b	1.0	NaN

Then we look at what data type the columns are:

```
[4]: df.dtypes
```

```
[4]: timestamp    object
username         object
temperature      float64
heartrate        int64
build            object
latest           float64
note             object
dtype: object
```

We can also display the values and their frequency, for example for the column note:

```
[5]: df.note.value_counts()
```

```
[5]: note
wake      16496
user      16416
interval  16274
sleep     16226
update    16213
test      16068
Name: count, dtype: int64
```

2. Remove all null values (including the indication n/a)

2.1 ...with `pandas.read_csv`

`pandas.read_csv` usually already filters out many values that it recognises as NA or NaN. Further values can be specified with `na_values`.

```
[6]: df = pd.read_csv(
    "https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/iot_example_
    ↪with_nulls.csv",
    na_values=["n/a"],
)
```

2.2 ...with `pandas.DataFrame.dropna`

Missing values can be deleted with `pandas.DataFrame.dropna`.

To analyse the extent of the deletions, we display the extent of the DataFrame before and after the deletion with `pandas.DataFrame.shape`:

```
[7]: df.shape
[7]: (146397, 7)
```

```
[8]: df2 = df.dropna()

df2.shape
[8]: (46116, 7)
```

So we would lose more than 2/3 of the records with `pandas.DataFrame.dropna`.

In the next experiment, we want to analyse whether whole rows or columns contain no data. Here, `how='all'` removes rows or columns that do not contain values; `axis=1` says that empty rows should be removed.

```
[9]: df.dropna(how="all", axis=1).shape
[9]: (146397, 7)
```

This, too, does not bring us any further.

2.3 Find all columns where all data is present

```
[10]: complete_columns = list(df.columns)
```

```
[11]: complete_columns
```

```
[11]: ['timestamp',
      'username',
      'temperature',
      'heartrate',
      'build',
      'latest',
      'note']
```

2.4 Find all columns where the most data is available

```
[12]: list(df.dropna(thresh=int(df.shape[0] * 0.9), axis=1).columns)
```

```
[12]: ['timestamp', 'username', 'heartrate']
```

thresh requires a certain number of NA values, in our case 90% before axis=1 lashes a column.

2.5 Find all columns where data is missing

With `pandas.DataFrame.isnull` we can find missing values and with `pandas.DataFrame.any` we find out if an element is valid, usually via a column.

```
[13]: incomplete_columns = list(df.columns[df.isnull().any()])
```

```
[14]: incomplete_columns
```

```
[14]: ['temperature', 'build', 'latest', 'note']
```

With `num_missing` we can now output the number of missing values per column:

```
[15]: for col in incomplete_columns:
      num_missing = df[df[col].isnull() == True].shape[0]
      print(f"number missing for column {col}: {num_missing}")
```

```
number missing for column temperature: 32357
number missing for column build: 32350
number missing for column latest: 32298
number missing for column note: 48704
```

We can also output these values as a percentage:

```
[16]: for col in incomplete_columns:
      percent_missing = df[df[col].isnull() == True].shape[0] / df.shape[0]
      print(f"percent missing for column {col}: {percent_missing}")
```

```
percent missing for column temperature: 0.22102228870810195
percent missing for column build: 0.22097447352063226
percent missing for column latest: 0.22061927498514314
percent missing for column note: 0.332684412931959
```

2.6 Replace missing data

To be able to check our changes in the latest column, we use `pandas.Series.value_counts`. The method returns a series containing the number of unique values:

```
[17]: df.latest.value_counts()
```

```
[17]: latest
0.0    75735
1.0    38364
Name: count, dtype: int64
```

Now we replace the missing values in the column latest with 0 with `DataFrame.fillna`:


```
[18]: df.latest = df.latest.fillna(0)
```

```
[19]: df.latest.value_counts()
```

```
[19]: latest
0.0    108033
1.0     38364
Name: count, dtype: int64
```

2.7 Replace missing data using backfill

To make the records follow each other in their chronological order, we first set the index for timestamp with `set_index`:

```
[20]: df = df.set_index("timestamp")
```

```
[21]: df.head(20)
```

```
[21]:
```

	username	temperature	heartrate	\
timestamp				
2017-01-01T12:00:23	michaelsmith	12.0	67	
2017-01-01T12:01:09	kharrison	6.0	78	
2017-01-01T12:01:34	smithadam	5.0	89	
2017-01-01T12:02:09	eddierodriguez	28.0	76	
2017-01-01T12:02:36	kenneth94	29.0	62	
2017-01-01T12:03:04	bryanttodd	13.0	86	
2017-01-01T12:03:51	andrea98	17.0	81	
2017-01-01T12:04:35	scott28	16.0	76	
2017-01-01T12:05:05	hillpamela	5.0	82	
2017-01-01T12:05:41	moorejeffrey	25.0	63	
2017-01-01T12:06:21	njohnson	NaN	63	
2017-01-01T12:06:53	gsutton	29.0	80	
2017-01-01T12:07:41	jessica48	22.0	83	
2017-01-01T12:08:08	hornjohn	16.0	73	
2017-01-01T12:08:35	gramirez	24.0	73	
2017-01-01T12:09:05	schmidtsamuel	NaN	78	
2017-01-01T12:09:48	derrick47	NaN	63	
2017-01-01T12:10:23	beckercharles	12.0	61	
2017-01-01T12:10:57	ipittman	11.0	69	
2017-01-01T12:11:34	sabrina65	22.0	82	

	build	latest	note
timestamp			
2017-01-01T12:00:23	4e6a7805-8faa-2768-6ef6-eb3198b483ac	0.0	interval
2017-01-01T12:01:09	7256b7b0-e502-f576-62ec-ed73533c9c84	0.0	wake
2017-01-01T12:01:34	9226c94b-bb4b-a6c8-8e02-cb42b53e9c90	0.0	NaN
2017-01-01T12:02:09	NaN	0.0	update
2017-01-01T12:02:36	122f1c6a-403c-2221-6ed1-b5caa08f11e0	0.0	NaN
2017-01-01T12:03:04	0897dbe5-9c5b-71ca-73a1-7586959ca198	0.0	interval
2017-01-01T12:03:51	1c07ab9b-5f66-137d-a74f-921a41001f4e	1.0	NaN
2017-01-01T12:04:35	7a60219f-6621-e548-180e-ca69624f9824	0.0	interval
2017-01-01T12:05:05	a8b87754-a162-da28-2527-4bce4b3d4191	1.0	NaN
2017-01-01T12:05:41	585f1a3c-0679-0ffe-9132-508933c70343	0.0	wake

(continues on next page)

(continued from previous page)

2017-01-01T12:06:21	e09b6001-125d-51cf-9c3f-9cb686c19d02	0.0	NaN
2017-01-01T12:06:53	607c9f6e-2bdf-a606-6d16-3004c6958436	1.0	update
2017-01-01T12:07:41	03e1a07b-3e14-412c-3a69-6b45bc79f81c	0.0	update
2017-01-01T12:08:08	NaN	0.0	interval
2017-01-01T12:08:35	NaN	0.0	wake
2017-01-01T12:09:05	b9890c1e-79d5-8979-63ae-6c08a4cd476a	0.0	NaN
2017-01-01T12:09:48	b60bd7de-4057-8a85-f806-e6eec1350338	0.0	interval
2017-01-01T12:10:23	b1dacc73-c8b7-1d7d-ee02-578da781a71e	0.0	test
2017-01-01T12:10:57	1aef7db8-9a3e-7dc9-d7a5-781ec0efd200	0.0	user
2017-01-01T12:11:34	8075d058-7dae-e2ec-d47e-58ec6d26899b	1.0	NaN

We then use `pandas.DataFrame.groupby` to group the records by username and then fill the missing data with the `backfill` method of `pandas.core.groupby.DataFrameGroupBy.fillna`. `limit` defines the maximum number of consecutive NaN values:

```
[22]: df.temperature = df.groupby("username").temperature.fillna(
        method="backfill", limit=3
    )
```

```
[23]: for col in incomplete_columns:
        num_missing = df[df[col].isnull() == True].shape[0]
        print(f"number missing for column {col}: {num_missing}")
```

```
number missing for column temperature: 22633
number missing for column build: 32350
number missing for column latest: 0
number missing for column note: 48704
```

Arguments of the function `fillna`:

Argument	Description
<code>value</code>	Scalar value or dict-like object used to fill in missing values.
<code>Method</code>	interpolation; by default <code>ffill</code> if the function is called without further arguments
<code>axis</code>	Axis to be filled; default <code>axis=0</code>
<code>inplace</code>	changes the calling object without creating a copy
<code>limit</code>	for padding in forward and backward direction, maximum number of consecutive periods to pad

4.1.2 Detecting and filtering outliers

Filtering or transforming outliers is largely a matter of applying array operations. Consider a `DataFrame` with some normally distributed data:

```
[1]: import numpy as np
import pandas as pd

df = pd.DataFrame(np.random.randn(1000, 4))

df.describe()
```

```
[1]:
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.022943	0.034915	0.018331	0.018969
std	1.046883	0.987310	0.972618	1.009193
min	-3.870700	-2.833648	-3.466318	-3.491794
25%	-0.664047	-0.621664	-0.603689	-0.671775
50%	0.012635	0.026107	0.028248	0.017063
75%	0.736662	0.697112	0.636774	0.743254
max	3.700035	3.006204	2.751574	3.405041

Suppose you want to find values in one of the columns whose absolute value is greater than 3:

```
[2]: col = df[1]

col[col.abs() > 3]
```

```
[2]: 365      3.006204
Name: 1, dtype: float64
```

To select all rows where value is greater than 3 or less than -3 in one of the columns, you can apply `pandas.DataFrame.any` to a Boolean DataFrame, using `any(axis=1)` to check if a value is in a row:

```
[3]: df[(df.abs() > 3).any(axis=1)]
```

```
[3]:
```

	0	1	2	3
67	-0.065879	1.783196	0.554033	-3.000936
123	0.246540	-0.588655	1.366174	-3.491794
209	-3.615275	-1.539901	-1.109978	1.557272
326	-3.543526	-0.123145	-1.166289	-0.793547
357	1.168551	-0.951635	-0.892777	3.405041
361	3.116807	-0.184181	0.694654	-1.116010
365	-0.274058	3.006204	0.638351	-0.117403
384	-3.006891	0.871370	-0.888511	-0.498219
388	1.104036	0.127207	1.306627	3.164983
504	-0.344477	1.190462	-3.466318	-1.577547
711	3.700035	0.449643	-0.130976	-0.231090
841	-3.870700	0.165213	-0.401433	1.267149
956	3.188822	-0.048598	0.921613	-0.281664
957	-0.326832	-0.324983	0.384806	-3.062165

On this basis, the values can be limited to an interval between -3 and 3. For this we use the instruction `np.sign(df)`, which generates values 1 and -1, depending on whether the values in `df` are positive or negative:

```
[4]: df[df.abs() > 3] = np.sign(df) * 3
```

```
df.describe()
```

```
[4]:
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.023974	0.034909	0.018797	0.018954
std	1.037146	0.987292	0.971056	1.005623
min	-3.000000	-2.833648	-3.000000	-3.000000
25%	-0.664047	-0.621664	-0.603689	-0.671775
50%	0.012635	0.026107	0.028248	0.017063

(continues on next page)

(continued from previous page)

75%	0.736662	0.697112	0.636774	0.743254
max	3.000000	3.000000	2.751574	3.000000

4.1.3 String comparisons

In this notebook we use the popular library for string comparisons [fuzzywuzzy](#). It is based on the built-in Python library [difflib](#). For more information on the various methods available and their differences, see the blog post [FuzzyWuzzy: Fuzzy String Matching in Python](#).

See also

- [textacy](#)

1. Installation

With *Spack* you can provide `fuzzywuzzy` and the optional `python-levenshtein` library in your kernel:

```
$ spack env activate python-311
$ spack install py-fuzzywuzzy+speedup
```

Alternatively, you can install the two libraries with other package managers, for example

```
$ pipenv install fuzzywuzzy[speedup]
```

2. Import

```
[1]: from fuzzywuzzy import fuzz, process
```

3. Example

```
[2]: berlin = [
    "Berlin, Germany",
    "Berlin, Deutschland",
    "Berlin",
    "Berlin, DE"]
```

String similarity

The similarity of the first two strings 'Berlin, Germany' and 'Berlin, Deutschland' seems low:

```
[3]: fuzz.ratio(berlin[0], berlin[1])
```

```
[3]: 65
```

Partial string similarity

Inconsistent partial strings are a common problem. To get around this, fuzzywuzzy uses a heuristic called *best partial*.

```
[4]: fuzz.partial_ratio(berlin[0], berlin[1])
```

```
[4]: 60
```

Token sorting

In token sorting, the string in question is given a token, the tokens are sorted alphabetically and then reassembled into a string, for example:

```
[5]: fuzz.ratio(berlin[1], berlin[2])
```

```
[5]: 48
```

```
[6]: fuzz.token_set_ratio(berlin[1], berlin[2])
```

```
[6]: 100
```

Further information

```
[7]: fuzz.ratio?
```

Extract from a list

```
[8]: choices = [
    "Germany",
    "Deutschland",
    "France",
    "United Kingdom",
    "Great Britain",
    "United States",
]
```

```
[9]: process.extract("DE", choices, limit=2)
```

```
[9]: [('Deutschland', 90), ('Germany', 45)]
```

```
[10]: process.extract("Vereinigtes Königreich", choices)
```

```
[10]: [('United Kingdom', 51),
      ('United States', 41),
      ('Germany', 39),
      ('Great Britain', 35),
      ('Deutschland', 31)]
```

```
[11]: process.extractOne("frankreich", choices)
```

```
[11]: ('France', 62)
```

```
[12]: process.extractOne("U.S.", choices)
[12]: ('United States', 86)
```

Known ports

FuzzyWuzzy is also ported to other languages! Here are some known ports:

- Java: `xpresso`
- Java: `xdrop fuzzywuzzy`
- Rust: `fuzzyrusty`
- JavaScript: `fuzzball.js`
- C++: `tmplt fuzzywuzzy`
- C#: `FuzzySharp`
- Go: `go-fuzzywuzzy`
- Pascal: `FuzzyWuzzy.pas`
- Kotlin: `FuzzyWuzzy-Kotlin`
- R: `fuzzywuzzyR`

4.1.4 Deduplicating data

In this notebook, we deduplicate data using the `Dedupe` library, which uses a flat neural network to learn from a little training.

See also

- `csvdedupe` offers a command line interface for `Dedupe`.

In addition, the same developers have created `parserator`, which you can use to extract text functions and train your own text extraction.

1. Load sample data

```
[1]: import pandas as pd

[2]: customers = pd.read_csv(
    "https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/customer_data_
    ↪duped.csv",
    encoding="utf-8",
)
```

2. Deduplicate with pandas

```
[3]: customers
```

```
[3]:
```

	name	job
0	Patricia Schaefer	Programmer, systems
1	Olivie Dubois	Ingénieur recherche et développement en agroal...
2	Mary Davies-Kirk	Public affairs consultant
3	Mirosława Eckbauer	Dispensing optician
4	Richard Bauer	Accountant, chartered certified
...
2075	Maurice Stey	Systems developer
2076	Linda Alexander	Commrcil horiculuri
2077	Diane Bailly	Pharmacien
2078	Jorge Riba Cerdán	Hotel manager
2079	Ryan Thompson	Brewing technologist

	company	street_address
0	Estrada-Best	398 Paul Drive
1	Moreno	rue Lucas Benard
2	Baker Ltd	Flat 3\nPugh mews
3	Ladeck GmbH	Mijo-Lübs-Straße 12
4	Hoffman-Rocha	6541 Rodriguez Wall
...
2075	Linke Margraf GmbH & Co. OHG	Laila-Scheibe-Allee 2/0
2076	Webb, Ballald and Vasquel	5594 Persn Ciff
2077	Voisin	527, rue Dijoux
2078	Amador-Diego	Rambla de Adriana Barceló 854 Puerta 3
2079	Smith-Sullivan	136 Rodriguez Point

	city	state	email
0	Christianview	Delaware	lambdavid@gmail.com
1	Saint Anastasie-les-Bains	AR	berthelotjacqueline@mahe.fr
2	Stanleyfurt	ZA	middletonconor@hotmail.com
3	Neubrandenburg	Berlin	sophia01@yahoo.de
4	Carlosmouth	Texas	tross@jensen-ware.org
...
2075	Luckenwalde	Hamburg	gutknechtevelyn@niemeier.com
2076	Mooneybury	Maryland	ahleythoa@aail.co
2077	Duval-les-Bains	CH	aruiz@reynaud.fr
2078	Huesca	Asturias	manuelamosquera@yahoo.com
2079	Bradfordborough	North Dakota	lcruz@gmail.com

	user_name
0	ndavidson
1	manonallain
2	colemanmichael
3	romanjunitz
4	adam78
...	...
2075	dkreusel
2076	kennethrchn
2077	dorothee41

(continues on next page)

(continued from previous page)

```
2078      eugenia17
2079      cnewton

[2080 rows x 8 columns]
```

2.2 Show data types

For this we use `pandas.DataFrame.dtypes`:

```
[4]: customers.dtypes
[4]: name          object
     job          object
     company       object
     street_address object
     city          object
     state         object
     email         object
     user_name     object
     dtype: object
```

2.3 Determine missing values

`pandas.isnull` shows for an array-like object whether values are missing:

- NaN in numeric arrays
- None or NaN in object arrays
- NaT in datetimelike

See also

- `notna` for the boolean inverse of `pandas.isna`
- `Series.isna` for the missing values in a series
- `DataFrame.isna` for the missing values in a DataFrame
- `Index.isna` for the missing values in an index

```
[5]: for col in customers.columns:
     print(col, customers[col].isnull().sum())

name 0
job 0
company 0
street_address 0
city 0
state 0
email 0
user_name 0
```


2.4 Determine duplicate records

```
[6]: customers.duplicated()
```

```
[6]: 0      False
      1      False
      2      False
      3      False
      4      False
      ...
      2075   False
      2076   False
      2077   False
      2078   False
      2079   False
      Length: 2080, dtype: bool
```

`customers.duplicated()` does not yet give us the desired indication of whether there are duplicate records. In the following, we will output all data records for which `True` is returned:

```
[7]: customers[customers.duplicated()]
```

```
[7]: Empty DataFrame
      Columns: [name, job, company, street_address, city, state, email, user_name]
      Index: []
```

Apparently there are no duplicated records.

2.5 Delete duplicated data

Deleting duplicated records with `drop_duplicates` should therefore not change anything and leave the number of records at 2080:

```
[8]: customers.drop_duplicates()
```

```
[8]:
```

	name	job	\
0	Patricia Schaefer	Programmer, systems	
1	Olivie Dubois	Ingénieur recherche et développement en agroal...	
2	Mary Davies-Kirk	Public affairs consultant	
3	Miroslaw Eckbauer	Dispensing optician	
4	Richard Bauer	Accountant, chartered certified	
...	
2075	Maurice Stey	Systems developer	
2076	Linda Alexander	Commrcil horiculuri	
2077	Diane Bailly	Pharmacien	
2078	Jorge Riba Cerdán	Hotel manager	
2079	Ryan Thompson	Brewing technologist	

	company	street_address	\
0	Estrada-Best	398 Paul Drive	
1	Moreno	rue Lucas Benard	
2	Baker Ltd	Flat 3\nPugh mews	
3	Ladeck GmbH	Mijo-Lübs-Straße 12	

(continues on next page)

(continued from previous page)

```

4             Hoffman-Rocha                6541 Rodriguez Wall
...
2075 Linke Margraf GmbH & Co. OHG          Laila-Scheibe-Allee 2/0
2076 Webb, Ballald and Vasquel            5594 Persn Ciff
2077                               Voisin      527, rue Dijoux
2078 Amador-Diego Rambla de Adriana Barceló 854 Puerta 3
2079 Smith-Sullivan                        136 Rodriguez Point

              city            state            email \
0      Christianview      Delaware      lambdavid@gmail.com
1      Saint Anastasie-les-Bains      AR      berthelotjacqueline@mahe.fr
2      Stanleyfurt        ZA      middletonconor@hotmail.com
3      Neubrandenburg      Berlin      sophia01@yahoo.de
4      Carlosmouth        Texas      tross@jensen-ware.org
...
2075 Luckenwalde          Hamburg      gutknechtevelyn@niemeier.com
2076 Mooneybury           Maryland      ahleythoa@aail.co
2077 Duval-les-Bains      CH      aruiz@reynaud.fr
2078 Huesca               Asturias      manuelamosquera@yahoo.com
2079 Bradfordborough     North Dakota      lcruz@gmail.com

      user_name
0      ndavidson
1      manonallain
2      colemanmichael
3      romanjunitz
4      adam78
...
2075      dkreusel
2076      kennethrchn
2077      dorothee41
2078      eugenia17
2079      cnewton

[2080 rows x 8 columns]

```

Now we want to delete only those records whose `user_name` is identical:

```
[9]: customers.drop_duplicates(["user_name"])
```

```

[9]:
      name            job \
0      Patricia Schaefer      Programmer, systems
1      Olivie Dubois      Ingénieur recherche et développement en agroal...
2      Mary Davies-Kirk      Public affairs consultant
3      Mirosława Eckbauer      Dispensing optician
4      Richard Bauer      Accountant, chartered certified
...
2074      Rhonda James      Recruitment consultant
2076      Linda Alexander      Commrcil horiculuri
2077      Diane Bailly      Pharmacien
2078      Jorge Riba Cerdán      Hotel manager
2079      Ryan Thompson      Brewing technologist

```

(continues on next page)

(continued from previous page)

```

      company                                street_address \
0      Estrada-Best                          398 Paul Drive
1      Moreno                               rue Lucas Benard
2      Baker Ltd                            Flat 3\nPugh mews
3      Ladeck GmbH                         Mijo-Lübs-Straße 12
4      Hoffman-Rocha                       6541 Rodriguez Wall
...
2074  Turner, Bradley and Scott             28382 Stokes Expressway
2076  Webb, Ballald and Vasquel            5594 Persn Ciff
2077      Voisin                           527, rue Dijoux
2078      Amador-Diego Rambla de Adriana Barceló 854 Puerta 3
2079      Smith-Sullivan                   136 Rodriguez Point

      city                                state                                email \
0      Christianview                     Delaware                      lambdavid@gmail.com
1      Saint Anastasie-les-Bains          AR                      berthelotjacqueline@mahe.fr
2      Stanleyfurt                       ZA                      middletonconor@hotmail.com
3      Neubrandenburg                    Berlin                      sophia01@yahoo.de
4      Carlosmouth                       Texas                      tross@jensen-ware.org
...
2074      Port Gabrielaport New Hampshire                      zroberts@hotmail.com
2076      Mooneybury                  Maryland                      ahleythoa@aail.co
2077      Duval-les-Bains              CH                      aruiz@reynaud.fr
2078      Huesca                      Asturias                      manuelamosquera@yahoo.com
2079      Bradfordborough North Dakota                      lcruz@gmail.com

      user_name
0      ndavidson
1      manonallain
2      colemanmichael
3      romanjunitz
4      adam78
...
2074      heathscott
2076      kennethrchn
2077      dorothee41
2078      eugenia17
2079      cnewton

[2029 rows x 8 columns]
```

This deleted 51 records.

3. dedupe

Alternatively, we can detect the duplicated data with the [Dedupe](#) library, which uses a flat neural network to learn from a small training.

See also

[csvdedupe](#) provides a command line tool for dedupe.

In addition, the same developers have created [parserator](#), which you can use to extract text functions and train your own text extraction.

3.1 Configure Dedupe

Now we define the fields to be taken care of during deduplication and create a new `deduper` object:

```
[10]: import os

import dedupe

customers = pd.read_csv(
    "https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/customer_data_
    ↪duped.csv",
    encoding="utf-8",
)

[11]: variables = [
    {"field": "name", "type": "String"},
    {"field": "job", "type": "String"},
    {"field": "company", "type": "String"},
    {"field": "street_address", "type": "String"},
    {"field": "city", "type": "String"},
    {"field": "state", "type": "String", "has_missing": True},
    {"field": "email", "type": "String", "has_missing": True},
    {"field": "user_name", "type": "String"},
]

deduper = dedupe.Dedupe(variables)
```

If the value of a field is missing, this missing value should be represented as a `None` object. However, by `'has_missing': True`, a new, additional field is created to indicate whether the data was present or not, and the missing data is given a null.

See also

- [Missing Data](#)

```
[12]: deduper
[12]: <dedupe.api.Dedupe at 0x7fd414e1a3a0>

[13]: customers.shape
[13]: (2080, 8)
```

4. Create training data

```
[14]: deduper.prepare_training(customers.T.to_dict())
```

`prepare_training` initialises active learning with our data and, optionally, with existing training data.

`T` mirrors the `DataFrame` across its diagonal by writing rows as columns and vice versa. For this, `pandas.DataFrame.transpose` is used.

5. Active learning

Use `dedupe.console_label` to train your dedupe instance. When Dedupe finds a record pair, you will be asked to label it as a duplicate. You can use the `y`, `n` and `u` keys to label duplicates. Press `f` when you are finished.

```
[15]: dedupe.console_label(deduper)
```

```
name : Frédérique Lejeune-Daniel
job : Technicien chimiste
company : Schmitt
street_address : chemin Denise Ferrand
city : Saint CharlotteVille
state : IE
email : jchretien@costa.com
user_name : joseph60
```

```
name : Frédérique Lejeune-Daniel
job : Tecce cse
company : Sctmitt
street_address : chemin Denise Ferrand
city : Saint ChalotteVille
state : IE
email : jchretien@costacom
user_name : joseph60
```

```
0/10 positive, 0/10 negative
Do these records refer to the same thing?
(y)es / (n)o / (u)nsure / (f)inished
```

```
y
```

```
name : Jose Carlos Pérez Arias
job : Engineer, maintenance (IT)
company : Marquez PLC
street_address : Pasadizo Ángel Sureda 715 Piso 3
city : La Rioja
state : Córdoba
email : cifuentesraquel@peralta.com
user_name : gonzalo63
```

```
name : Jose Carlos Pérez Arias
job : Egieer, maiteace (IT)
company : Marquez PLC
street_address : Psdizo Ángel Sured 715 Piso
city : La Rioja
```

(continues on next page)

(continued from previous page)

```
state : Córdoba
email : ifuenteraque@perata.om
user_name : gonzalo6
```

```
1/10 positive, 0/10 negative
Do these records refer to the same thing?
(y)es / (n)o / (u)nsure / (f)inished / (p)revious
```

```
y
```

```
name : Julio Agustín Amaya
job : Tax adviser
company : Piñol, Belmonte and Codina
street_address : Callejón de Gregorio Bustamante 28 Piso 7
city : Las Palmas
state : Salamanca
email : usolana@jáuregui-pedraza.com
user_name : gloriaolmo
```

```
name : Julio Agustín Amaya
job : Tax adviser
company : Piñolk Belmonke and Codina
street_address : Calleón de Gregorio Bustamante 28 Piso 7
city : La Pala
state : Salamanca
email : usolana@jáuregui-pedraza.om
user_name : gloriaolmo
```

```
2/10 positive, 0/10 negative
Do these records refer to the same thing?
(y)es / (n)o / (u)nsure / (f)inished / (p)revious
```

```
y
```

```
name : Monique Marty
job : Maoqiie
company : Arnfud
street_address : 70, rue de Carre
city : CheallierBour
state : EC
email : frederiquerichard@cohen.com
user_name : marquesseastie
```

```
name : Monique Marty
job : Maroquinier
company : Arnaud
street_address : 70, rue de Carre
city : ChevallierBourg
state : EC
email : frederiquerichard@cohen.com
user_name : marquessebastien
```

```
3/10 positive, 0/10 negative
Do these records refer to the same thing?
```

(continues on next page)

(continued from previous page)

(y)es / (n)o / (u)nsure / (f)inished / (p)revious

y

```

name : Susan Aubry
job : Direeur d'gee bire
company : Payet George2 S2A2S2
street_address : , rue Inè Valentr
city : Nicolas
state : FI
email : milletedith@sf.f
user_name : tthierry

```

```

name : Susan Aubry
job : Directeur d'agence bancaire
company : Payet Georges S.A.S.
street_address : 67, rue Inès Valentin
city : Nicolas
state : FI
email : milletedith@sfr.fr
user_name : tthierry

```

4/10 positive, 0/10 negative

Do these records refer to the same thing?

(y)es / (n)o / (u)nsure / (f)inished / (p)revious

f

Finished labeling

The last training dataset compared make it clear that we did not delete this duplicate with our `drop_duplicates` example above - `marquesseastie` and `marquessebastien` were recognised as different.

`Dedupe.train` adds the record pairs you marked to the training data and updates the matching model.

With `index_predicates=True`, deduplication also takes into account predicates based on the indexing of the data.

When you are done, save your training data with `Dedupe.write_settings`.

```

[16]: settings_file = "csv_example_learned_settings"
if os.path.exists(settings_file):
    print("reading from", settings_file)
    with open(settings_file, "rb") as f:
        deduper = dedupe.StaticDedupe(f)
else:
    deduper.train(index_predicates=True)
    with open(settings_file, "wb") as sf:
        deduper.write_settings(sf)

```

With `dedupe.Dedupe.partition`, records that all refer to the same entity are identified and returned as tuples that are a sequence of record IDs and confidence values. For more details on the confidence value, see `dedupe.Dedupe.cluster`.

```

[17]: dupes = deduper.partition(customers.T.to_dict())

```

```
[18]: dupes
```

```
[18]: (((84, 1600), (1.0, 1.0)),
      ((136, 1360), (1.0, 1.0)),
      ((670, 1170), (1.0, 1.0)),
      ((856, 1781), (1.0, 1.0)),
      ((902, 942), (1.0, 1.0)),
      ((1395, 1560), (1.0, 1.0)),
      ((1594, 1706), (1.0, 1.0)),
      ((0,), (1.0,)),
      ((1,), (1.0,)),
      ...]
```

We can also output only individual entries:

```
[19]: dupes[1]
```

```
[19]: ((136, 1360), (1.0, 1.0))
```

We can then display these with `pandas.DataFrame.iloc`:

```
[20]: customers.iloc[[136,1360]]
```

```
[20]:
```

	name	job	company \
136	Frédérique Lejeune-Daniel	Technicien chimiste	Schmitt
1360	Frédérique Lejeune-Daniel	Tecce cse	Sctmitt

	street_address	city	state	email \
136	chemin Denise Ferrand	Saint Charlotte	Ville IE	jchretien@costa.com
1360	chemin Denise Ferrand	Saint Chalotte	Ville IE	jchretien@costacom

	user_name
136	joseph60
1360	joseph60

4.1.5 pandas DataFrame Validation with Bulwark

Bulwark is a package for property-based testing of pandas dataframes. The project was heavily influenced by the no longer supported **Engarde** library.

1. Installation

```
$ pipenv install bulwark
Installing bulwark...
Adding bulwark to Pipfile's [packages]...
✓ Installation Succeeded
Locking [dev-packages] dependencies...
✓ Success!
Updated Pipfile.lock (0d075a)!
```


2. Use

2.1 Checks

With the `bulwark.checks` module you can check many common assumptions, e.g.

- `has_columns` checks whether certain columns exist in such-and-such a way and in the correct order
- `has_dtypes` checks the data types of columns
- `has_no_infs` checks if there are no `numpy.inf` in the DataFrame
- `has_no_nans` checks if there are no `numpy.nan` in the DataFrame
- `has_set_within_vals` checks if the values specified in a dict are a subset of the associated column
- `has_unique_index` checks if the index is unique
- `is_monotonic` checks whether values of a column are ascending or descending
- `one_to_many` checks whether there is an n:1 relationship between two columns

The checks are then very simple, e.g. the check whether there are no `numpy.nan` in the column pipe with

```
import bulwark.checks as ck

df.pipe(ck.has_no_nans())
```

2.2 Decorators

For each check, bulwark creates `decorators`, e.g. `@dc.IsShape((-1, 10))` or `@dc.IsMonotonic(strict=True)`.

CustomCheck

You can also create your own custom functions, for example:

```
[1]: import bulwark.checks as ck
import bulwark.decorators as dc
import numpy as np
import pandas as pd

def len_longer_than(df, l):
    if len(df) <= l:
        raise AssertionError("df is not as long as expected.")
    return df

@dc.CustomCheck(len_longer_than, 10)
def append_a_df(df, df2):
    return pd.concat([df, df2], ignore_index=True)

df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
df2 = pd.DataFrame({"a": [1, np.nan, 3, 4], "b": [4, 5, 6, 7]})
```

(continues on next page)

(continued from previous page)

```
append_a_df(df, df2)
```

```
-----
AssertionError                                Traceback (most recent call last)
Cell In[1], line 21
    18 df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
    19 df2 = pd.DataFrame({"a": [1, np.nan, 3, 4], "b": [4, 5, 6, 7]})
--> 21 append_a_df(df, df2)

File ~/local/share/venv/python-311-6zxVKbDJ/lib/python3.11/site-packages/bulwark/
decorators.py:81, in CustomCheck.__call__.<locals>.decorated(*args, **kwargs)
    78 df = f(*args, **kwargs)
    79 if self.enabled:
    80     # differs from BaseDecorator
--> 81     ck.custom_check(df, self.check_func, **self.check_func_params)
    82 return df

File ~/local/share/venv/python-311-6zxVKbDJ/lib/python3.11/site-packages/bulwark/
checks.py:588, in custom_check(df, check_func, *args, **kwargs)
    576 """Assert that `check(df, *args, **kwargs)` is true.
    577
    578 Args:
    (...)
    585
    586 """
    587 try:
--> 588     check_func(df, *args, **kwargs)
    589 except AssertionError as e:
    590     msg = "{} is not true.".format(check_func.__name__)

Cell In[1], line 9, in len_longer_than(df, l)
     7 def len_longer_than(df, l):
     8     if len(df) <= l:
--> 9         raise AssertionError("df is not as long as expected.")
    10     return df

AssertionError: len_longer_than is not true.
```

MultiCheck

With MultiCheck you can run several tests at the same time and see all the errors at once, for example:

```
[2]: @dc.MultiCheck(
      checks={
          ck.has_no_nans: {"columns": None},
          len_longer_than: {"l": 6}
      },
      warn=False,
  )
  def append_a_df(df, df2):
```

(continues on next page)

(continued from previous page)

```

return pd.concat([df, df2], ignore_index=True)

df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
df2 = pd.DataFrame({"a": [1, np.nan, 3, 4], "b": [4, 5, 6, 7]})

append_a_df(df, df2)
-----
AssertionError                                Traceback (most recent call last)
Cell In[2], line 15
     12 df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
     13 df2 = pd.DataFrame({"a": [1, np.nan, 3, 4], "b": [4, 5, 6, 7]})
--> 15 append_a_df(df, df2)

File ~/local/share/venv/python-311-6zxVKbDJ/lib/python3.11/site-packages/bulwark/
decorators.py:24, in BaseDecorator.__call__.<locals>.decorated(*args, **kwargs)
     22 df = f(*args, **kwargs)
     23 if self.enabled:
--> 24     self.check_func(df, **self.check_func_params)
     25 return df

File ~/local/share/venv/python-311-6zxVKbDJ/lib/python3.11/site-packages/bulwark/
checks.py:570, in multi_check(df, checks, warn)
     568     return df
     569 elif error_msgs:
--> 570     raise AssertionError("\n".join(str(i) for i in error_msgs))
     572 return df

AssertionError: (4, 'a')

```

4.1.6 Hypothesis: Property-based testing

In this notebook we use property-based testing to find problems in our code. [Hypothesis](#) is a library similar to Haskell's [Quickcheck](#). We'll get to know it in more detail later, along with other test libraries: [Hypothesis](#). [Hypothesis](#) can also provide mock objects and tests for numpy data types.

1. Imports

```
[1]: import re

from hypothesis import assume, given
from hypothesis.strategies import emails, integers, tuples
```

2. Find range

```
[2]: def calculate_range(tuple_obj):
      return max(tuple_obj) - min(tuple_obj)
```

3. Test with strategies and given

With `hypothesis.strategies` you can create different test data. For this, Hypothesis provides strategies for most types and arguments restrict the possibilities to suit your needs. In the example below, we use the `integers` strategy, which is applied to the function with the `Python-Decorator` `@given`. More specifically, it takes our test function and converts it into a parameterised one to run over wide ranges of matching data:

```
[3]: @given(tuples(integers(), integers(), integers()))
def test_calculate_range(tup):
    result = calculate_range(tup)
    assert isinstance(result, int)
    assert result > 0
```

```
[4]: test_calculate_range()
```

```
-----
AssertionError                                Traceback (most recent call last)
Cell In[4], line 1
----> 1 test_calculate_range()

Cell In[3], line 2, in test_calculate_range()
      1 @given(tuples(integers(), integers(), integers()))
----> 2 def test_calculate_range(tup):
      3     result = calculate_range(tup)
      4     assert isinstance(result, int)

[... skipping hidden 1 frame]

Cell In[3], line 5, in test_calculate_range(tup)
      3 result = calculate_range(tup)
      4 assert isinstance(result, int)
----> 5 assert result > 0

AssertionError:
Falsifying example: test_calculate_range(
    tup=(0, 0, 0),
)
```

Now we correct the test with `>=` and check it again:

```
[5]: @given(tuples(integers(), integers()))
def test_calculate_range(tup):
    result = calculate_range(tup)
    assert isinstance(result, int)
    assert result >= 0
```

```
[6]: test_calculate_range()
```

3. Check against regular expressions

Regular expressions can be used to check strings for certain syntactical rules. In Python, you can use `re.match` to check regular expressions.

Note

On the website [regex101](https://regex101.com/) you can first try out your regular expressions.

As an example, let's try to find out the username and the domain from email addresses:

```
[7]: def parse_email(email):
    result = re.match(
        "(?P<username>\w+).(?P<domain>[\w\.\.]+)",
        email,
    ).groups()
    return result
```

Now we write a test `test_parse_email` to check our method. As input values we use the `emails` strategy of Hypothesis. As result we expect for example:

```
('0', 'A.com')
('F', 'j.EeHNqsx')
...
```

In the test, we assume on the one hand that two entries are always returned and that a dot (.) occurs in the second entry.

```
[8]: @given(emails())
def test_parse_email(email):
    result = parse_email(email)
    # print(result)
    assert len(result) == 2
    assert "." in result[1]
```

```
[9]: test_parse_email()
```

```
-----
AttributeError                                Traceback (most recent call last)
Cell In[9], line 1
----> 1 test_parse_email()

Cell In[8], line 2, in test_parse_email()
      1 @given(emails())
----> 2 def test_parse_email(email):
      3     result = parse_email(email)
```

(continues on next page)

(continued from previous page)

```

4     # print(result)

[... skipping hidden 1 frame]

Cell In[8], line 3, in test_parse_email(email)
      1 @given(emails())
      2 def test_parse_email(email):
----> 3     result = parse_email(email)
      4     # print(result)
      5     assert len(result) == 2

Cell In[7], line 5, in parse_email(email)
      1 def parse_email(email):
      2     result = re.match(
      3         "(?P<username>[w+]).(?P<domain>[w\\.]+)",
      4         email,
----> 5     ).groups()
      6     return result

AttributeError: 'NoneType' object has no attribute 'groups'
Falsifying example: test_parse_email(
    email='=@A.ac',
)

```

With Hypothesis, two examples were found that make it clear that our regular expression in the `parse_email` method is not yet sufficient: `0/0@A.ac` and `/@A.ac`. After we have adapted our regular expression accordingly, we can call the test again:

```

[10]: def parse_email(email):
      result = re.match(
          "(?P<username>[\\.\\w\\-\\!~#$$%&|{}\\+\\/^\\`\\'=\\*']+).(P<domain>[\\w\\.\\-]+)",
          email,
      ).groups()
      return result

[11]: test_parse_email()

```

4.1.7 TDDA: Test-Driven Data Analysis

TDDA uses file inputs (such as NumPy arrays or Pandas DataFrames) and a set of constraints that are stored as a JSON file.

- **Reference Tests** supports the creation of reference tests based on either unittest or pytest.
- **Constraints** is used to retrieve constraints from a (pandas) DataFrame, write them out as JSON and check whether records satisfy the constraints in the constraints file. It also supports tables in a variety of relational databases.
- **Rexpy** is a tool for automatically deriving regular expressions from a column in a pandas DataFrame or from a (Python) list of examples.

1. Imports

```
[1]: import numpy as np
import pandas as pd

from tdda.constraints import discover_df, verify_df

[2]: df = pd.read_csv(
    "https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/iot_example.csv"
)
```

2. Check data

With `pandas.DataFrame.sample` we display ten random data sets:

```
[3]: df.sample(10)
```

	timestamp	username	temperature	heartrate	\
34835	2017-01-15T10:01:00	jwilliams	6	89	
120063	2017-02-18T11:25:06	ericliu	28	77	
130072	2017-02-22T11:12:22	lee99	9	79	
79530	2017-02-02T07:23:17	jamie46	5	82	
53587	2017-01-22T22:02:38	daniellemacias	26	63	
112089	2017-02-15T07:04:50	carlosyoung	23	63	
91218	2017-02-06T23:19:13	gutierreznathan	11	72	
105807	2017-02-12T18:57:19	gutierrezashley	28	72	
51928	2017-01-22T05:58:03	uschwartz	29	81	
70436	2017-01-29T15:52:55	crystalunderwood	23	82	

	build	latest	note
34835	8aee9cf7-84c6-6935-22ff-9b034d9aa1f4	0	update
120063	c1179197-abcc-ee64-2851-cb2ed21baa1a	0	wake
130072	184d2848-9367-71cb-be72-6bbd57074857	0	NaN
79530	1c56c556-2ba0-11fb-5a27-29289487b748	1	wake
53587	acd9a855-077c-dda7-c73f-7621f3179f17	0	sleep
112089	71dfc6de-147e-00f1-da89-6e4489a33aba	0	user
91218	166e2a51-ae08-bd1f-3cee-3c65a0d5740b	0	NaN
105807	9b1984e4-a620-68f6-c639-2db7774fe27c	0	user
51928	54bf638e-68e1-9514-02df-acbc5417443a	0	user
70436	8f003e04-509d-e412-0979-0c9f9526f1e8	1	NaN

And with `pandas.DataFrame.dtypes` we display the data types for the individual columns:

```
[4]: df.dtypes

[4]: timestamp    object
username        object
temperature      int64
heartrate       int64
build           object
latest          int64
```

(continues on next page)

(continued from previous page)

```
note          object
dtype: object
```

3. Creating a constraints object

With `discover_constraints` a constraints object can be created.

```
[5]: constraints = discover_df(df)

[6]: constraints
[6]: <tda.constraints.base.DatasetConstraints at 0x161ac2c50>

[7]: constraints.fields
[7]: Fields([('timestamp', <tda.constraints.base.FieldConstraints at 0x161ac1350>),
            ('username', <tda.constraints.base.FieldConstraints at 0x161ac1650>),
            ('temperature',
             <tda.constraints.base.FieldConstraints at 0x136bc2790>),
            ('heartrate', <tda.constraints.base.FieldConstraints at 0x161ac1f10>),
            ('build', <tda.constraints.base.FieldConstraints at 0x161ac2350>),
            ('latest', <tda.constraints.base.FieldConstraints at 0x161ac2990>),
            ('note', <tda.constraints.base.FieldConstraints at 0x161ac2d50>)])
```

4. Writing the constraints into a file

```
[8]: with open("../data/ignore-iot_constraints.tdda", "w") as f:
      f.write(constraints.to_json())
```

If we take a closer look at the file, we can see that, for example, a string with 19 characters is expected for the `timestamp` column and temperature expects integers with values from 5-29.

```
[9]: !cat ../data/ignore-iot_constraints.tdda
{
  "creation_metadata": {
    "local_time": "2023-07-26 18:18:37",
    "utc_time": "2023-07-26 16:16:37",
    "creator": "TDDA 2.0.09",
    "host": "fay.local",
    "user": "veit",
    "n_records": 146397,
    "n_selected": 146397
  },
  "fields": {
    "timestamp": {
      "type": "string",
      "min_length": 19,
      "max_length": 19,
      "max_nulls": 0,
```

(continues on next page)

(continued from previous page)

```

        "no_duplicates": true
    },
    "username": {
        "type": "string",
        "min_length": 3,
        "max_length": 21,
        "max_nulls": 0
    },
    "temperature": {
        "type": "int",
        "min": 5,
        "max": 29,
        "sign": "positive",
        "max_nulls": 0
    },
    "heartrate": {
        "type": "int",
        "min": 60,
        "max": 89,
        "sign": "positive",
        "max_nulls": 0
    },
    "build": {
        "type": "string",
        "min_length": 36,
        "max_length": 36,
        "max_nulls": 0,
        "no_duplicates": true
    },
    "latest": {
        "type": "int",
        "min": 0,
        "max": 1,
        "sign": "non-negative",
        "max_nulls": 0
    },
    "note": {
        "type": "string",
        "min_length": 4,
        "max_length": 8,
        "allowed_values": [
            "interval",
            "sleep",
            "test",
            "update",
            "user",
            "wake"
        ]
    }
}

```

5. Checking data frames

To do this, we first read in a new csv file with pandas and then have ten data records output as examples:

```
[10]: new_df = pd.read_csv(
    "https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/iot_example_
    ↪with_nulls.csv"
)
new_df.sample(10)
```

```
[10]:
```

	timestamp	username	temperature	heartrate	\
985	2017-01-01T21:31:32	qmartinez	9.0	85	
102620	2017-02-11T12:30:15	laurenwilliams	19.0	85	
59859	2017-01-25T10:06:55	xwright	22.0	66	
110018	2017-02-14T11:13:49	wibarra	NaN	68	
1736	2017-01-02T04:47:11	joshuaperez	NaN	79	
85078	2017-02-04T12:37:57	oaustin	23.0	63	
145979	2017-02-28T20:00:11	hholder	21.0	77	
37789	2017-01-16T14:12:06	kennethdavis	8.0	62	
114849	2017-02-16T09:27:34	bobby64	20.0	67	
107275	2017-02-13T09:05:58	okirby	29.0	64	

	build	latest	note
985	76f24b19-9d9e-17b8-4d02-c6e6f00e9f13	1.0	NaN
102620	5ea80d05-56b2-5632-8480-cf3ef40d34a4	0.0	user
59859	NaN	0.0	test
110018	9d04e55f-c9e1-2ab9-666b-9de0f739431a	0.0	test
1736	3d320343-34f9-bf79-ec39-aaafa061c39c	1.0	NaN
85078	ccf940cd-9b59-e444-1a68-5f7f7590d5db	0.0	NaN
145979	NaN	0.0	NaN
37789	9bc9bc15-bf2a-7098-bf85-73fa42e30df2	0.0	user
114849	2c5dc517-d725-bc66-7d6d-25716025476a	0.0	sleep
107275	NaN	1.0	interval

We see several fields that are output as NaN. Now, to analyse this systematically, we apply `verify_df` to our new DataFrame. Here, `passes` returns the number of passed constraints, and `failures` returns the number of failed constraints.

```
[11]: v = verify_df(new_df, '../data/ignore-iot_constraints.tdda')
```

```
[12]: v
```

```
[12]: <tdda.constraints.pd.constraints.PandasVerification at 0x1636e7450>
```

```
[13]: v.passes
```

```
[13]: 30
```

```
[14]: v.failures
```

```
[14]: 3
```

We can also display which constraints passed and failed in which columns:

```
[15]: print(str(v))
```

FIELDS:

```
timestamp: 0 failures 5 passes type min_length max_length max_nulls no_
↳ duplicates
username: 0 failures 4 passes type min_length max_length max_nulls
temperature: 1 failure 4 passes type min max sign max_nulls
heartrate: 0 failures 5 passes type min max sign max_nulls
build: 1 failure 4 passes type min_length max_length max_nulls no_duplicates
latest: 1 failure 4 passes type min max sign max_nulls
note: 0 failures 4 passes type min_length max_length allowed_values
```

SUMMARY:

```
Constraints passing: 30
Constraints failing: 3
```

Alternatively, we can also display these results in tabular form:

```
[16]: v.to_frame()
```

```
[16]:
```

	field	failures	passes	type	min	min_length	max	max_length	\
0	timestamp	0	5	True	NaN	True	NaN	True	
1	username	0	4	True	NaN	True	NaN	True	
2	temperature	1	4	True	True	NaN	True	NaN	
3	heartrate	0	5	True	True	NaN	True	NaN	
4	build	1	4	True	NaN	True	NaN	True	
5	latest	1	4	True	True	NaN	True	NaN	
6	note	0	4	True	NaN	True	NaN	True	

	sign	max_nulls	no_duplicates	allowed_values
0	NaN	True	True	NaN
1	NaN	True	NaN	NaN
2	True	False	NaN	NaN
3	True	True	NaN	NaN
4	NaN	False	True	NaN
5	True	False	NaN	NaN
6	NaN	NaN	NaN	True

4.1.8 Data validation with Voluptuous (schema definitions)

In this notebook we use [Voluptuous](#) to define schemas for our data. We can then use schema checking at various points in our cleanup to ensure that we meet the criteria. Finally, we can use schema checking exceptions to flag, set aside or remove impure or invalid data.

See also

- [Validr](#)
- [marshmallow](#)

1. Imports

```
[1]: import logging

from datetime import datetime

import pandas as pd

from voluptuous import ALLOW_EXTRA, All, Range, Required, Schema
from voluptuous.error import Invalid, MultipleInvalid
```

- `Required` marks the node of a schema as required and optionally specifies a default value, see also [voluptuous.schema_builder.Required](#).
- `Range` limits the value to a range where either `min` or `max` can be omitted; see also [voluptuous.validators.Range](#).
- `ALL` is used for cross-field validations: checks the basic structure of the data in a first pass and only in the second pass the cross-field validation is applied; see also [voluptuous.validators.All](#).
- `ALLOW_EXTRA` allows additional dictionary keys.
- `MultipleInvalid` is based on `Invalid`, see also [voluptuous.error.MultipleInvalid](#).
- `Invalid` marks data as invalid, see also [voluptuous.error.Invalid](#).

2. Logger

```
[2]: logger = logging.getLogger(0)
logger.setLevel(logging.WARNING)
```

3. Read sample data

```
[3]: sales = pd.read_csv(
    "https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/sales_data.csv"
)
```

4. Examine data

```
[4]: sales.head()
```

```
[4]: Unnamed: 0      timestamp      city  store_id  sale_number  \
0          0  2018-09-10 05:00:45  Williamburgh          6        1530
1          1  2018-09-12 10:01:27    Ibarraberg          1        2744
2          2  2018-09-13 12:01:48    Sarachester          2        1908
3          3  2018-09-14 20:02:19  Caldwellbury         14         771
4          4  2018-09-16 01:03:21    Erikaland          11        1571

      sale_amount      associate
0         1167.0      Gary Lee
1          258.0    Daniel Davis
2          266.0    Michael Roth
3         -108.0  Michaela Stewart
4         -372.0    Mark Taylor
```

```
[5]: sales.shape
```

```
[5]: (213, 7)
```

```
[6]: sales.dtypes
```

```
[6]: Unnamed: 0      int64
timestamp      object
city           object
store_id       int64
sale_number    int64
sale_amount    float64
associate      object
dtype: object
```

5. Define schema

In the column `sale_amount` all values should be between 2.5 and 1450.99:

```
[7]: schema = Schema(
    {
        Required("sale_amount"): All(float, Range(min=2.50, max=1450.99)),
    },
    extra=ALLOW_EXTRA,
)
```

To be able to use the elements of one column as keys and the elements of another column as values, we simply make the desired column the index of the DataFrame and transpose it with the function `.T()`; see also [pandas.DataFrame.transpose](#).

```
[8]: error_count = 0
for s_id, sale in sales.T.to_dict().items():
    try:
        schema(sale)
    except MultipleInvalid as e:
```

(continues on next page)

(continued from previous page)

```

logging.warning(
    "issue with sale: %s (%s) - %s", s_id, sale["sale_amount"], e
)
error_count += 1

```

```

WARNING:root:issue with sale: 3 (-108.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 4 (-372.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 5 (-399.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 6 (-304.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 7 (-295.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 10 (-89.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 13 (-303.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 15 (-432.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 19 (-177.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 20 (-154.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 22 (-130.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 23 (1487.0) - value must be at most 1450.99 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 25 (-145.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 28 (1471.0) - value must be at most 1450.99 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 31 (-259.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 38 (-241.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 40 (-4.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 41 (1581.0) - value must be at most 1450.99 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 45 (1529.0) - value must be at most 1450.99 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 46 (-238.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 48 (-284.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 51 (-164.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 55 (-184.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 56 (-304.0) - value must be at least 2.5 for dictionary_

```

(continues on next page)

(continued from previous page)

```

↪value @ data['sale_amount']
WARNING:root:issue with sale: 59 (1579.0) - value must be at most 1450.99 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 60 (-455.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 63 (1551.0) - value must be at most 1450.99 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 65 (-397.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 69 (-400.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 70 (1482.0) - value must be at most 1450.99 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 71 (-321.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 74 (-47.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 76 (-68.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 86 (1454.0) - value must be at most 1450.99 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 101 (-213.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 103 (-144.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 104 (-265.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 107 (-349.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 111 (-78.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 112 (-310.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 116 (1570.0) - value must be at most 1450.99 for↪
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 120 (1490.0) - value must be at most 1450.99 for↪
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 123 (-179.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 124 (-391.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 129 (1504.0) - value must be at most 1450.99 for↪
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 130 (-91.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 132 (-372.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 141 (1512.0) - value must be at most 1450.99 for↪
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 142 (-449.0) - value must be at least 2.5 for dictionary↪
↪value @ data['sale_amount']
WARNING:root:issue with sale: 149 (1494.0) - value must be at most 1450.99 for↪

```

(continues on next page)

(continued from previous page)

```

↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 152 (-405.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 155 (1599.0) - value must be at most 1450.99 for_
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 156 (1527.0) - value must be at most 1450.99 for_
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 157 (-462.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 162 (-358.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 164 (-78.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 167 (-358.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 171 (-391.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 178 (-304.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 180 (-9.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 187 (1475.0) - value must be at most 1450.99 for_
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 194 (-433.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 195 (-329.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 196 (-147.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 203 (-319.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 206 (-132.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 207 (-20.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']
WARNING:root:issue with sale: 209 (1539.0) - value must be at most 1450.99 for_
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 211 (-167.0) - value must be at least 2.5 for dictionary_
↪value @ data['sale_amount']

```

```
[9]: error_count
```

```
[9]: 69
```

Currently, however, we do not yet know whether

- we have a wrongly defined schema
- possibly negative values are returned or incorrectly marked
- higher values are combined purchases or special sales

6. Adding a custom validation

```
[10]: def ValidDate(fmt="%Y-%m-%d %H:%M:%S"):
      return lambda v: datetime.strptime(v, fmt)
```

```
[11]: schema = Schema(
      {
          Required("timestamp"): All(ValidDate()),
      },
      extra=ALLOW_EXTRA,
  )
```

```
[12]: error_count = 0
      for s_id, sale in sales.T.to_dict().items():
          try:
              schema(sale)
          except MultipleInvalid as e:
              logging.warning(
                  "issue with sale: %s (%s) - %s", s_id, sale["timestamp"], e
              )
              error_count += 1
```

```
[13]: error_count
```

```
[13]: 0
```

7. Valid date structures are not yet valid dates

```
[14]: def ValidDate(fmt="%Y-%m-%d %H:%M:%S"):
      def validation_func(v):
          try:
              assert datetime.strptime(v, fmt) <= datetime.now()
          except AssertionError:
              raise Invalid("date is in the future! %s" % v)

      return validation_func
```

```
[15]: schema = Schema(
      {
          Required("timestamp"): All(ValidDate()),
      },
      extra=ALLOW_EXTRA,
  )
```

```
[16]: error_count = 0
      for s_id, sale in sales.T.to_dict().items():
          try:
              schema(sale)
          except MultipleInvalid as e:
              logging.warning(
```

(continues on next page)

(continued from previous page)

```
        "issue with sale: %s (%s) - %s", s_id, sale["timestamp"], e
    )
    error_count += 1
```

```
[17]: error_count
```

```
[17]: 0
```

4.1.9 Normalisation and Preprocessing

`sklearn.preprocessing` can be used in many ways to clean data:

- Standardisation with `StandardScaler`, `MinMaxScaler`, `MaxAbsScaler` or `RobustScaler`.
- Centring of kernel matrices with `KernelCenterer`.
- Non-linear transformations with `QuantileTransformer`, `PowerTransformer`
- Normalisation with `normalize`.
- Encoding of categorical features with `OrdinalEncoder`, `OneHotEncoder`.
- Discretisation (also known as quantisation or binning) with `KBinsDiscretizer`.
- Binarisation of features with `Binarizer`
- Imputation of missing values with `SimpleImputer`, `IterativeImputer` or `KNNImputer` where the added values can be marked with `MissingIndicator`.

See also

- `statsmodels`

Example

In the following example, we fill in mean values and do some scaling:

1. Imports

```
[1]: from datetime import datetime

import numpy as np
import pandas as pd

from sklearn import preprocessing
from sklearn.impute import SimpleImputer
```

```
[2]: hvac = pd.read_csv(
    "https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/HVAC_with_
    ↪nulls.csv"
)
```

2. Check data quality

Display data types with `pandas.DataFrame.dtypes`:

```
[3]: hvac.dtypes
```

```
[3]: Date          object
     Time          object
     TargetTemp    float64
     ActualTemp    int64
     System        int64
     SystemAge     float64
     BuildingID    int64
     10            float64
     dtype: object
```

Return dimensions of the DataFrame as a tuple with `pandas.DataFrame.shape`:

```
[4]: hvac.shape
```

```
[4]: (8000, 8)
```

Return first n rows with `pandas.DataFrame.head`:

```
[5]: hvac.head()
```

```
[5]:   Date      Time  TargetTemp  ActualTemp  System  SystemAge  BuildingID  10
0  6/1/13  0:00:01         66.0          58      13         20.0          4  NaN
1  6/2/13  1:00:01          NaN          68       3         20.0         17  NaN
2  6/3/13  2:00:01         70.0          73      17         20.0         18  NaN
3  6/4/13  3:00:01         67.0          63       2          NaN         15  NaN
4  6/5/13  4:00:01         68.0          74      16          9.0          3  NaN
```

3. Attribute the mean value to missing values

For this we use the mean strategy of `sklearn.impute.SimpleImputer`:

```
[6]: imp = SimpleImputer(missing_values=np.nan, strategy="mean")
```

```
[7]: hvac_numeric = hvac[["TargetTemp", "SystemAge"]]
```

```
[8]: imp = imp.fit(hvac_numeric.loc[:10])
```

For more information on `fit`, see the [Scikit Learn documentation](#).

`fit_transform` then transforms the adapted data:

```
[9]: transformed = imp.fit_transform(hvac_numeric)
```

```
[10]: transformed
```

```
[10]: array([[66.          , 20.          ],
          [67.50773481, 20.          ],
          [70.          , 20.          ],
```

(continues on next page)

(continued from previous page)

```
...,
[67.50773481, 4.      ],
[65.      , 23.      ],
[66.      , 21.      ]])
```

```
[11]: hvac["TargetTemp"], hvac["SystemAge"] = transformed[:, 0], transformed[:, 1]
```

Now we display the first rows with the changed data records:

```
[12]: hvac.head()
```

```
[12]:
```

	Date	Time	TargetTemp	ActualTemp	System	SystemAge	BuildingID	10
0	6/1/13	0:00:01	66.0000000	58	13	20.0000000	4	NaN
1	6/2/13	1:00:01	67.507735	68	3	20.0000000	17	NaN
2	6/3/13	2:00:01	70.0000000	73	17	20.0000000	18	NaN
3	6/4/13	3:00:01	67.0000000	63	2	15.386643	15	NaN
4	6/5/13	4:00:01	68.0000000	74	16	9.0000000	3	NaN

4. Scale

To standardise data sets that look like standard normally distributed data, we can use `sklearn.preprocessing.scale`. This can be used to determine the factors by which a value increases or decreases. We can use this to scale the current temperature.

```
[13]: hvac["ScaledTemp"] = preprocessing.scale(hvac["ActualTemp"])
```

```
[14]: hvac["ScaledTemp"].head()
```

```
[14]: 0    -1.293272
      1     0.048732
      2     0.719733
      3    -0.622270
      4     0.853934
      Name: ScaledTemp, dtype: float64
```

`sklearn.preprocessing.MinMaxScaler` scales the terms so that they lie between a certain minimum and maximum value, often between zero and one. This has the advantage of making the scaling more robust to very small standard deviations of features.

```
[15]: min_max_scaler = preprocessing.MinMaxScaler()
```

```
[16]: temp_minmax = min_max_scaler.fit_transform(hvac[["ActualTemp"]])
```

```
[17]: temp_minmax
```

```
[17]: array([[0.12],
           [0.52],
           [0.72],
           ...,
           [0.56],
           [0.32],
           [0.44]])
```

Now we also add `temp_minmax` as a new column:

```
[18]: hvac["MinMaxScaledTemp"] = temp_minmax[:,0]
hvac["MinMaxScaledTemp"].head()
```

```
[18]: 0    0.12
      1    0.52
      2    0.72
      3    0.32
      4    0.76
      Name: MinMaxScaledTemp, dtype: float64
```

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

```
[19]:   Date      Time  TargetTemp  ActualTemp  System  SystemAge  BuildingID  10  \
0  6/1/13  0:00:01   66.000000         58      13   20.000000         4  NaN
1  6/2/13  1:00:01   67.507735         68       3   20.000000        17  NaN
2  6/3/13  2:00:01   70.000000         73      17   20.000000        18  NaN
3  6/4/13  3:00:01   67.000000         63       2   15.386643        15  NaN
4  6/5/13  4:00:01   68.000000         74      16    9.000000         3  NaN

      ScaledTemp  MinMaxScaledTemp
0   -1.293272         0.12
1    0.048732         0.52
2    0.719733         0.72
3   -0.622270         0.32
4    0.853934         0.76
```

4.1.10 Assigning satellite data to geo-locations

Example: Tracking the International Space Station with Dask

In this notebook we will use two APIs:

1. Google Maps Geocoder
2. Open Notify API for ISS location

We will use them to track ISS location and next transit time with respect to a list of cities. To create our charts and parallelise data intelligently, we will use Dask, specifically *Dask Delayed*.

1. Imports

```
[1]: import logging
import sys

from datetime import datetime
from math import radians
from operator import itemgetter
from time import sleep

import numpy as np
import requests
```

(continues on next page)

(continued from previous page)

```
from dask import delayed
from sklearn.metrics import DistanceMetric
```

2. Logger

```
[2]: logger = logging.getLogger()
     logger.setLevel(logging.INFO)
```

3. Latitude and longitude pairs from a list of cities

See also

- [Location APIs](#)

```
[3]: def get_lat_long(address):
     resp = requests.get(
         "https://eu1.locationiq.org/v1/search.php",
         params={"key": "92e7ba84cf3465", "q": address, "format": "json"},
     )
     if resp.status_code != 200:
         print("There was a problem with your request!")
         print(resp.content)
         return
     data = resp.json()[0]
     return {
         "name": data.get("display_name"),
         "lat": float(data.get("lat")),
         "long": float(data.get("lon")),
     }
```

```
[4]: get_lat_long("Berlin, Germany")
```

```
[4]: {'name': 'Berlin, 10117, Germany', 'lat': 52.5170365, 'long': 13.3888599}
```

```
[5]: locations = []
     for city in [
         "Seattle, Washington",
         "Miami, Florida",
         "Berlin, Germany",
         "Singapore",
         "Wellington, New Zealand",
         "Beirut, Lebanon",
         "Beijing, China",
         "Nairobi, Kenya",
         "Cape Town, South Africa",
         "Buenos Aires, Argentina",
     ]:
         locations.append(get_lat_long(city))
         sleep(2)
```

```
[6]: locations
[6]: [{'name': 'Seattle, King County, Washington, USA',
      'lat': 47.6038321,
      'long': -122.3300624},
      {'name': 'Miami, Miami-Dade County, Florida, USA',
      'lat': 25.7741728,
      'long': -80.19362},
      {'name': 'Berlin, 10117, Germany', 'lat': 52.5170365, 'long': 13.3888599},
      {'name': 'Singapore', 'lat': 1.357107, 'long': 103.8194992},
      {'name': 'Wellington, Wellington City, Wellington, 6011, New Zealand',
      'lat': -41.2887953,
      'long': 174.7772114},
      {'name': 'Beirut, Beirut Governorate, Lebanon',
      'lat': 33.8959203,
      'long': 35.47843},
      {'name': 'Beijing, Dongcheng District, Beijing, 100010, China',
      'lat': 39.906217,
      'long': 116.3912757},
      {'name': 'Nairobi, Kenya', 'lat': -1.2832533, 'long': 36.8172449},
      {'name': 'Cape Town, City of Cape Town, Western Cape, 8001, South Africa',
      'lat': -33.928992,
      'long': 18.417396},
      {'name': 'Autonomous City of Buenos Aires, Comuna 6, Autonomous City of Buenos Aires, ↵
      ↪Argentina',
      'lat': -34.6075682,
      'long': -58.4370894}]
```

4. Retrieve ISS data and determine transit times of cities

```
[7]: def get_spaceship_location():
      resp = requests.get("http://api.open-notify.org/iss-now.json")
      location = resp.json()["iss_position"]
      return {
          "lat": float(location.get("latitude")),
          "long": float(location.get("longitude")),
      }

[8]: def great_circle_dist(lon1, lat1, lon2, lat2):
      dist = DistanceMetric.get_metric("haversine")
      lon1, lat1, lon2, lat2 = map(np.radians, [lon1, lat1, lon2, lat2])

      X = [[lat1, lon1], [lat2, lon2]]
      kms = 6367
      return (kms * dist.pairwise(X)).max()

[9]: def iss_dist_from_loc(issloc, loc):
      distance = great_circle_dist(
          issloc.get("long"), issloc.get("lat"), loc.get("long"), loc.get("lat")
      )
      logging.info("ISS is ~%dkm from %s", int(distance), loc.get("name"))
```

(continues on next page)

(continued from previous page)

```
return distance
```

```
[10]: def iss_pass_near_loc(loc):
      resp = requests.get(
          "http://api.open-notify.org/iss-pass.json",
          params={"lat": loc.get("lat"), "lon": loc.get("lon")},
      )
      data = resp.json().get("response")[0]
      td = datetime.fromtimestamp(data.get("risetime")) - datetime.now()
      m, s = divmod(int(td.total_seconds()), 60)
      h, m = divmod(m, 60)
      logging.info(
          "ISS will pass near %s in %02d:%02d:%02d", loc.get("name"), h, m, s
      )
      return td.total_seconds()
```

```
[11]: iss_dist_from_loc(get_spaceship_location(), locations[2])
```

```
INFO:root:ISS is ~12639km from Berlin, 10117, Germany
```

```
[11]: 12639.759939298825
```

```
[12]: iss_pass_near_loc(locations[2])
```

```
INFO:root:ISS will pass near Berlin, 10117, Germany in 00:25:14
```

```
[12]: 1514.253889
```

5. Creating a delayed pipeline

```
[13]: output = []
```

```
for loc in locations:
    issloc = delayed(get_spaceship_location)()
    dist = delayed(iss_dist_from_loc)(issloc, loc)
    output.append((loc.get("name"), dist))

closest = delayed(lambda x: sorted(x, key=itemgetter(1))[0])(output)
```

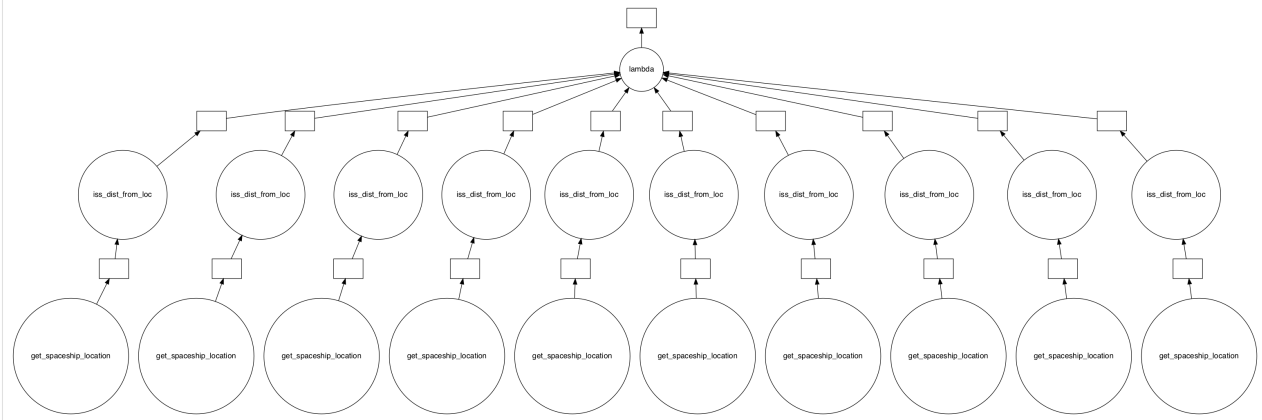
```
[14]: closest
```

```
[14]: Delayed('lambda-5ab5a78f-cb72-4168-bce1-f9983fdb8a2e')
```


6. Show DAG

```
[15]: closest.visualize()
```

```
[15]:
```



7. compute()

```
[16]: closest.compute()
```

```
INFO:root:ISS is ~4685km from Miami, Miami-Dade County, Florida, USA
INFO:root:ISS is ~15205km from Beirut, Beirut Governorate, Lebanon
INFO:root:ISS is ~5919km from Seattle, King County, Washington, USA
INFO:root:ISS is ~6279km from Autonomous City of Buenos Aires, Comuna 6, Autonomous City
↳ of Buenos Aires, Argentina
INFO:root:ISS is ~12625km from Berlin, 10117, Germany
INFO:root:ISS is ~13137km from Cape Town, City of Cape Town, Western Cape, 8001, South
↳ Africa
INFO:root:ISS is ~16194km from Singapore
INFO:root:ISS is ~16298km from Nairobi, Kenya
INFO:root:ISS is ~13905km from Beijing, Dongcheng District, Beijing, 100010, China
INFO:root:ISS is ~8405km from Wellington, Wellington City, Wellington, 6011, New Zealand
```

```
[16]: ('Miami, Miami-Dade County, Florida, USA', 4685.887400314564)
```


VISUALISE DATA

We have outsourced the visualisation of data to a separate tutorial: [PyViz Tutorial](#).

PERFORMANCE

Python can be used to write and test code quickly because it is an interpreted language that types dynamically. However, these are also the reasons it is slow when performing simple tasks repeatedly, for example in loops.

When developing code, there can often be trade-offs between different implementations. However, at the beginning of the development of an algorithm, it is usually counterproductive to worry about the efficiency of the code.

«We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.»¹

6.1 k-Means example

In the following, I show examples of the [k-means algorithm](#) to form a previously known number of groups from a set of objects. This can be achieved in the following three steps:

1. Choose the first *k* elements as cluster centres
2. Assign each new element to the cluster with the least increase in variance.
3. Update the cluster centre

Steps 2 and 3 are repeated until the assignments no longer change.

A possible implementation with pure Python could look like this:

Listing 1: `py_kmeans.py`

```
# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

def dist(x, y):
    """Calculate the distance"""
    return sum((xi - yi) ** 2 for xi, yi in zip(x, y))

def find_labels(points, centers):
    """Assign points to a cluster."""
    labels = []
    for point in points:
        distances = [dist(point, center) for center in centers]
```

(continues on next page)

¹ Donald Knuth, founder of [Literate Programming](#), in *Computer Programming as an Art* (1974)

(continued from previous page)

```

        labels.append(distances.index(min(distances)))
    return labels

def compute_centers(points, labels):
    """Calculate the cluster centres."""
    n_centers = len(set(labels))
    n_dims = len(points[0])

    centers = [[0 for i in range(n_dims)] for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]

    for label, point in zip(labels, points):
        counts[label] += 1
        centers[label] = [a + b for a, b in zip(centers[label], point)]

    return [[x / count for x in center] for center, count in zip(centers, counts)]

def kmeans(points, n_clusters):
    """Calculates the cluster centres repeatedly until nothing changes."""
    centers = points[-n_clusters:].tolist()
    while True:
        old_centers = centers
        labels = find_labels(points, centers)
        centers = compute_centers(points, labels)
        if centers == old_centers:
            break
    return labels

```

We can create sample data with:

```

from sklearn.datasets import make_blobs

points, labels_true = make_blobs(
    n_samples=1000, centers=3, random_state=0, cluster_std=0.60
)

```

And finally, we can perform the calculation with:

```

kmeans(points, 10)

```

6.2 Performance measurements

Once you have worked with your code, it can be useful to examine its efficiency more closely. The *iPython Profiler* or *scalene* can be used for this.

See also:

- [airspeed velocity \(asv\)](#) is a tool for benchmarking Python packages during their lifetime. Runtime, memory consumption and even user-defined values can be recorded and displayed in an interactive web frontend.
- [Python Speed Center](#)
- [Tracing the Python GIL](#)

6.2.1 iPython Profiler

IPython provides access to a wide range of functions to measure times and create profiles. The following magic IPython commands are explained here:

Command	Description
<code>%time</code>	Time to execute a single statement
<code>%timeit</code>	Average time it took to execute a single statement repeatedly
<code>%prun</code>	Run code with the profiler
<code>%lprun</code>	Run code with the line-by-line profiler
<code>%memit</code>	Measure the memory usage of a single statement
<code>%mprun</code>	Executes the code with the line-by-line memory profiler

The last four commands are not contained in IPython itself, but in the modules [line_profiler](#) and [memory_profiler](#).

See also

- [Penn Machine Learning Benchmarks](#)

`%timeit` and `%time`

We saw the `%timeit` line and `%%timeit` cell magic in the introduction of the magic functions in IPython magic commands. They can be used to measure the timing of the repeated execution of code snippets:

```
[1]: %timeit sum(range(100))
```

321 ns ± 1.6 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)

Note that `%timeit` executes the execution multiple times in a loop. If the number of loops is not specified with `-n`, `%timeit` automatically adjusts the number so that sufficient measurement accuracy is achieved:

```
[2]: %%timeit
total = 0
for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
```

99.1 ms ± 310 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

Sometimes repeating an operation is not the best option, for example when we have a list that we want to sort. Here we may be misled by repeated surgery. Sorting a presorted list is much faster than sorting an unsorted list, so repeating it distorts the result:

```
[3]: import random
```

```
L = [random.random() for i in range(100000)]  
%timeit L.sort()
```

```
205 µs ± 4.34 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Then the `%time` function might be a better choice. `%time` should also be the better choice for long-running commands, when short system-related delays are unlikely to affect the result:

```
[4]: import random
```

```
L = [random.random() for i in range(100000)]  
%time L.sort()
```

```
CPU times: user 10.4 ms, sys: 302 µs, total: 10.7 ms  
Wall time: 10.6 ms
```

Sorting an already sorted list:

```
[5]: %time L.sort()
```

```
CPU times: user 373 µs, sys: 5 µs, total: 378 µs  
Wall time: 379 µs
```

Note how much faster the pre-sorted list is to be sorted, but also note how much longer the timing with `%time` takes compared to `%timeit`, even for the pre-sorted list. This is due to the fact that `%timeit` is doing some clever things to keep system calls from interfering with the timing. This prevents, for example, the garbage collection of Python objects that are no longer used and that could otherwise affect the time measurement. Because of this, the `%timeit` results are usually noticeably faster than the `%time` results.

Profiling for scripts: `%prun`

A program is made up of many individual instructions, and sometimes it is more important to measure those instructions in context than to measure them yourself. Python includes a built-in [Code-Profiler](#). However, IPython offers a much more convenient way to use this profiler in the form of the magic function `%prun`.

As an example, let's define a simple function that does some calculations:

```
[6]: def sum_of_lists(N):  
    total = 0  
    for i in range(5):  
        L = [j ^ (j >> i) for j in range(N)]  
        total += sum(L)  
    return total
```

```
[7]: %prun sum_of_lists(1000000)
```


In the notebook the output looks something like this:

```
14 function calls in 9.597 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
5      8.121    1.624    8.121    1.624 <ipython-input-15-f105717832a2>:4(
↳<listcomp>)
5      0.747    0.149    0.747    0.149 {built-in method builtins.sum}
1      0.665    0.665    9.533    9.533 <ipython-input-15-f105717832a2>:1(sum_of_
↳lists)
1      0.065    0.065    9.597    9.597 <string>:1(<module>)
1      0.000    0.000    9.597    9.597 {built-in method builtins.exec}
1      0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'
↳objects}
```

The result is a table that shows the execution time for each function call, sorted by total time. In this case, most of the time is consumed with list comprehension within `sum_of_lists`. This gives us clues as to where we could improve the efficiency of the algorithm.

Profiling line by line: `%lprun`

Profiling with `%prun` is useful, but sometimes a line-by-line profile report is more insightful. This isn't built into Python or IPython, but there is a package available, [line_profiler](#), that enables this. This can be provided in your kernel with

```
$ spack env activate python-374
$ spack install py-line-profiler ^python@3.7.4%gcc@9.1.0
```

Alternatively, you can install `line_profiler` with other package managers, for example

```
$ pipenv install line_profiler
```

Now you can load IPython with the `line_profiler` extension:

```
[8]: %load_ext line_profiler
```

The `%lprun` command profiles each function line by line. In this case, you must explicitly specify which functions are of interest for creating the profile:

```
[9]: %lprun -f sum_of_lists sum_of_lists(5000)
```

The result looks something like this:

```
Timer unit: 1e-06 s

Total time: 0.015145 s
File: <ipython-input-6-f105717832a2>
Function: sum_of_lists at line 1

Line #      Hits          Time  Per Hit   % Time  Line Contents
=====
      1              1          1.0      1.0      0.0      def sum_of_lists(N):
      2              1          1.0      1.0      0.0          total = 0
```

(continues on next page)

(continued from previous page)

3	6	11.0	1.8	0.1	for i in range(5):
4	5	14804.0	2960.8	97.7	L = [j ^ (j >> i) for j in
↪range(N)]					
5	5	329.0	65.8	2.2	total += sum(L)
6	1	0.0	0.0	0.0	return total

The time is given in microseconds and we can see which line the function spends most of its time on. We may then be able to modify the script in such a way that the efficiency of the function can be increased.

More information about `%lprun` and the available options can be found in the IPython help function `%lprun?`.

Create a storage profile: `%memit` and `%mprun`

Another aspect of profiling is the amount of memory that an operation uses. This can be evaluated with another IPython extension, the `memory_profiler`. This can be provided in your kernel with

```
$ spack env activate python-311
$ spack install py-memory-profiler
```

Alternatively you can install `memory-profiler` with other package managers, for example

```
$ pipenv install memory_profiler
```

```
[10]: %load_ext memory_profiler
```

```
[11]: %memit sum_of_lists(1000000)
```

```
peak memory: 176.50 MiB, increment: 76.33 MiB
```

We see that this feature occupies approximately 100 MB of memory.

For a line-by-line description of memory usage, we can use the `%mprun` magic. Unfortunately, this magic only works for functions that are defined in separate modules and not for the notebook itself. So we first use the `%%file` magic to create a simple module called `mprun_demo.py` that contains our `sum_of_lists` function.

```
[12]: %%file mprun_demo.py
from memory_profiler import profile
```

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a
```

```
Writing mprun_demo.py
```

```
[13]: from mprun_demo import my_func
%mprun -f my_func my_func()
```

```
Filename: /Users/veit/cusy/trn/Python4DataScience/docs/performance/mprun_demo.py
```

Line #	Mem usage	Increment	Occurrences	Line Contents
--------	-----------	-----------	-------------	---------------

(continues on next page)

(continued from previous page)

=====				
3	123.8 MiB	123.8 MiB	1	@profile
4				def my_func():
5	131.5 MiB	7.7 MiB	1	a = [1] * (10 ** 6)
6	284.1 MiB	152.6 MiB	1	b = [2] * (2 * 10 ** 7)
7	284.1 MiB	0.0 MiB	1	del b
8	284.1 MiB	0.0 MiB	1	return a

Here the **Increment** column shows how much each row affects the total memory consumption: Note that when we calculate `b` we need about 160 MB of additional memory; however, this is not released again by deleting `b`.

More information about `%memit`, `%mprun` and their options can be found in the IPython help with `%memit?`.

pyheatmagic

`pyheatmagic` is an extension that allows the IPython magic command `%%heat` to display Python code as a heatmap with `Py-Heat`.

It can be easily installed in the kernel with

```
$ pipenv install py-heat-magic Installing py-heat-magic... ...
```

Loading the extension in IPython

```
[14]: %load_ext heat
```

Display the heat map

```
[15]: %%heat
```

```
def powfun(a, b):
    """Method to raise a to power b using pow() function."""
    return pow(a, b)

def powop(a, b):
    """Method to raise a to power b using ** operator."""
    return a**b

def powmodexp(a, b):
    """Method to raise a to power b using modular exponentiation."""
    base = a
    res = 1
    while b > 0:
        if b & 1:
```

(continues on next page)

(continued from previous page)

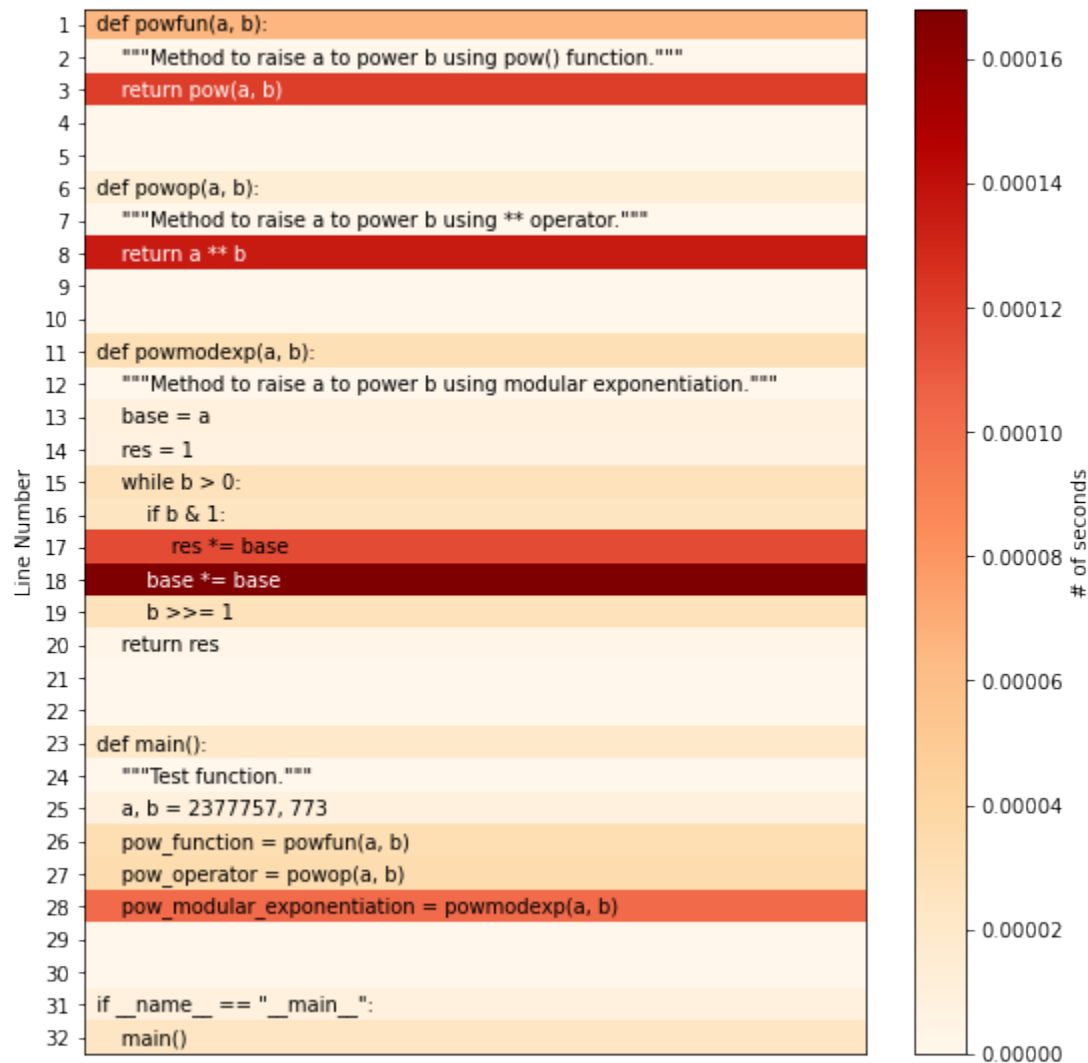
```

        res *= base
        base *= base
        b >>= 1
    return res

def main():
    """Test function."""
    a, b = 2377757, 773
    pow_function = powfun(a, b)
    pow_operator = powop(a, b)
    pow_modular_exponentiation = powmodexp(a, b)

if __name__ == "__main__":
    main()

```



Alternatively, the heatmap can also be saved as a file, for example with

```
%%heat -o pow-heatmap.png
```

6.2.2 scalene

scalene creates profiles for CPU and memory very quickly. The overhead is usually very low at 10–20%.

See also

- [GitHub](#)
- [PyPI](#)
- [scalene-paper.pdf](#)

Installation

Linux, MacOS and WSL:

```
$ pipenv install scalene
```

Use

1. An example programme for profiling

```
[1]: import numpy as np

def profile_me():
    for i in range(6):
        x = np.array(range(10**7))
        y = np.array(np.random.uniform(0, 100, size=(10**8)))
```

2. Load scalene

```
[2]: %load_ext scalene
```

Scalene extension successfully loaded. Note: Scalene currently only supports CPU+GPU profiling inside Jupyter notebooks. For full Scalene profiling, use the command line version.

NOTE: in Jupyter notebook on MacOS, Scalene cannot profile child processes. Do not run to try Scalene with multiprocessing in Jupyter Notebook.

3. Profile with only one line of code

```
[ ]: %scrunch profile_me()
import numpy as np

def profile_me():
```

(continues on next page)

(continued from previous page)

```

for i in range(6):
    x = np.array(range(10**7))
    y = np.array(np.random.uniform(0, 100, size=(10**8)))

```

Create a reduced profile (only rows with non-zero counts)

```

[ ]: %scrunch --reduced-profile profile_me()
import numpy as np

def profile_me():
    for i in range(6):
        x = np.array(range(10**7))
        y = np.array(np.random.uniform(0, 100, size=(10**8)))

```

For a complete list of options, contact:

```

[5]: %scrunch --help

usage: scalene [-h] [--version] [--column-width COLUMN_WIDTH]
              [--outfile OUTFILE] [--html] [--json] [--cli] [--stacks]
              [--web] [--viewer] [--reduced-profile]
              [--profile-interval PROFILE_INTERVAL] [--cpu] [--cpu-only]
              [--gpu] [--memory] [--profile-all]
              [--profile-only PROFILE_ONLY]
              [--profile-exclude PROFILE_EXCLUDE] [--use-virtual-time]
              [--cpu-percent-threshold CPU_PERCENT_THRESHOLD]
              [--cpu-sampling-rate CPU_SAMPLING_RATE]
              [--allocation-sampling-window ALLOCATION_SAMPLING_WINDOW]
              [--malloc-threshold MALLOC_THRESHOLD]
              [--program-path PROGRAM_PATH] [--memory-leak-detector]
              [--on | --off]

Scalene: a high-precision CPU and memory profiler, version 1.5.23 (2023.07.26)
]8;id=916670;https://github.com/plasma-umass/scalene\
↪https://github.com/plasma-umass/scalene]8;;\

command-line:
  % scalene [options] your_program.py [--- --your_program_args]
or
  % python3 -m scalene [options] your_program.py [--- --your_program_args]

in Jupyter, line mode:
  %scrunch [options] statement

in Jupyter, cell mode:
  %%scalene [options]
  your code here

options:
  -h, --help          show this help message and exit
  --version           prints the version number for this release of Scalene and exits

```

(continues on next page)

(continued from previous page)

```

--column-width COLUMN_WIDTH
    Column width for profile output (default: 132)
--outfile OUTFILE
    file to hold profiler output (default: stdout)
--html
    output as HTML (default: web)
--json
    output as JSON (default: web)
--cli
    forces use of the command-line
--stacks
    collect stack traces
--web
    opens a web tab to view the profile (saved as 'profile.html')
--viewer
    only opens the web UI (https://plasma-umass.org/scalene-gui/)
--reduced-profile
    generate a reduced profile, with non-zero lines only (default:
↪ False)
--profile-interval PROFILE_INTERVAL
    output profiles every so many seconds (default: inf)
--cpu
    profile CPU time (default: True )
--cpu-only
    profile CPU time (deprecated: use --cpu )
--gpu
    profile GPU time and memory (default: False )
--memory
    profile memory (default: True )
--profile-all
    profile all executed code, not just the target program (default:
↪ only the target program)
--profile-only PROFILE_ONLY
    profile only code in filenames that contain the given strings,
↪ separated by commas
(default: no restrictions)
--profile-exclude PROFILE_EXCLUDE
    do not profile code in filenames that contain the given strings,
↪ separated by commas
(default: no restrictions)
--use-virtual-time
    measure only CPU time, not time spent in I/O or blocking
↪ (default: False)
--cpu-percent-threshold CPU_PERCENT_THRESHOLD
    only report profiles with at least this percent of CPU time
↪ (default: 1%)
--cpu-sampling-rate CPU_SAMPLING_RATE
    CPU sampling rate (default: every 0.01s)
--allocation-sampling-window ALLOCATION_SAMPLING_WINDOW
    Allocation sampling window size, in bytes (default: 1048576
↪ bytes)
--malloc-threshold MALLOC_THRESHOLD
    only report profiles with at least this many allocations
↪ (default: 100)
--program-path PROGRAM_PATH
    The directory containing the code to profile (default: the path
↪ to the profiled program)
--memory-leak-detector
    EXPERIMENTAL: report likely memory leaks (default: True)
--on
    start with profiling on (default)
--off
    start with profiling off

```

When running Scalene in the background, you can suspend/resume profiling for the process ID that Scalene reports. For example:

```
% python3 -m scalene yourprogram.py &
```

(continues on next page)

(continued from previous page)

```
Scalene now profiling process 12345
to suspend profiling: python3 -m scalene.profile --off --pid 12345
to resume profiling:  python3 -m scalene.profile --on  --pid 12345
```

Profile with more than one line of code in a cell

```
[ ]: %%scalene --reduced-profile
x = 0
for i in range(1000):
    for j in range(1000):
        x += 1
```

6.3 Search for existing implementations

You should not try to reinvent the wheel: If there are existing implementations, you should use them. There are even two implementations for the k-means algorithm:

- `sklearn.cluster.KMeans`

```
from sklearn.cluster import KMeans

KMeans(10).fit_predict(points)
```

- `dask_ml.cluster.KMeans`

```
from dask_ml.cluster import KMeans

KMeans(10).fit(points).predict(points)
```

The best that could be said against these existing solutions is that they could create a considerable overhead in your project if you are not already using `scikit-learn` or `Dask-ML` elsewhere. In the following, I will therefore show you further possibilities to optimise your own code.

6.4 Find anti-patterns

Then you can use *perflint* to search your code for the most common performance anti-patterns in Python.

6.4.1 perflint

`perflint` is an extension for `pylint` for performance anti-patterns, among others:

W8101: unnecessary-list-cast

Unnecessary use of `list()` on an already iterable type.

W8102: incorrect-dictionary-iterator

Incorrect iterator method for dict: Python dictionaries store keys and values in two separate tables. They can be iterated separately. Using `.items()` and discarding either the key or the value with `_` is inefficient when `.keys()` or `.values()` can be used instead.

W8201: loop-invariant-statement

The loop is examined to determine statements or expressions whose result is constant on each iteration of a loop because they are based on named variables that are not changed during the iteration.

W8202: loop-global-usage

Global name usage in a loop: loading global variables is slower than loading local variables. The difference is marginal, but when passed in a loop, there can be a noticeable speed improvement.

R8203: loop-try-except-usage

Up until Python 3.10, `try...except` blocks are very computationally intensive compared to `if` statements.

Avoid using them in a loop as they can cause significant overhead. Refactor your code so that no iteration-specific details are required and put the entire loop in the `try` block.

W8204: memoryview-over-bytes

Slicing byte objects in loops is inefficient because it creates a copy of the data. Use `memoryview()` instead.

See also:

- [Zero-copy interactions](#)
- [Memoryview Benchmarks](#)
- [Memoryview Benchmarks 2](#)

W8205: dotted-import-in-loop

Direct import of the name `%s` is more efficient in a loop. In Python, you can import a module and then access submodules as attributes. You can also access functions as attributes of that module. This keeps the import statements to a minimum. However, if you use this method in a loop, it is inefficient because each loop pass loads the global, then the attribute, then the method.

W8301: use-tuple-over-list

Use a tuple instead of a list for an immutable sequence: both the construction and indexing of a tuple is faster than that of a list.

W8401: use-list-comprehension

Use list comprehensions with or without an `if` statement instead of a `for` loop.

W8402: use-list-copy

Use `list.copy()` instead of a `for` loop.

W8403: use-dict-comprehension

Uses a dictionary comprehension instead of a simple `for` loop.

See also:

- [Effective Python](#)

6.5 Vectorisations with NumPy

NumPy moves repetitive operations into a statically typed compiled layer, combining the fast development time of Python with the fast execution time of C. You may be able to use *Universal functions (ufunc)*, *vectorisation* and *Indexing and slicing* in all combinations to move repetitive operations into compiled code to avoid slow loops.

With NumPy we can do without some loops:

Listing 2: np_kmeans.py

```
# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

import numpy as np

def find_labels(points, centers):
```

The advantages of NumPy are that the Python overhead only occurs per array and not per array element. However, because NumPy uses a specific language for array operations, it also requires a different mindset when writing code. Finally, the batch operations can also lead to excessive memory consumption.

6.6 Special data structures

pandas

for SQL-like *Group operations* and
Aggregation.

This way you can also bypass the loop in the `compute_centers` method:

Listing 3: pd_kmeans.py

```
#
# SPDX-License-Identifier: BSD-3-Clause

diff = points[:, None, :] - centers
distances = (diff**2).sum(-1)
return distances.argmin(1)
```

scipy.spatial

for spatial queries like distances, nearest neighbours, k-Means etc (et cetera).

Our `find_labels` method can then be written more specifically:

Listing 4: sp_kmeans.py

```
import pandas as pd
from scipy.spatial import cKDTree
```

scipy.sparse

sparse matrices for 2-dimensional structured data.

Sparse

for N-dimensional structured data.

scipy.sparse.csgraph

for graph-like problems, for example searching for shortest paths.

Xarray

for grouping across multiple dimensions.

6.6.1 Parallelise pandas

In [Enhancing performance](#), some possibilities are described for improving the performance of pandas. However, there are also special libraries that can parallelise the processing of data frames.

cuDF

cuDF is a GPU DataFrame library that implements a [pandas-like API](#).

See also:

- [Docs](#)
- [GitHub](#)
- [PyPI](#)
- [Example notebooks](#)

Modin

Modin parallelises almost the entire Pandas API. In most cases, the existing Pandas code only needs to be extended by the following import:

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

The restrictions refer to `pd.read_json`, which is only implemented for `lines=True`.

See also:

- [Docs](#)
- [GitHub](#)

Dask

Dask `DataFrame` is a large parallel DataFrame made up of multiple pandas DataFrames. Here, the `dask.dataframe` API is a subset of the pandas API, although there are minor changes.

See also:

- [Home](#)
- [API docs](#)
- [Example notebook](#)

- [Tutorial](#)

6.7 Select compiler

6.7.1 Faster Cpython

At PyCon US in May 2021, Guido van Rossum presented [Faster CPython](#), a project that aims to double the speed of Python 3.11. The cooperation with the other Python core developers is regulated in [PEP 659 – Specializing Adaptive Interpreter](#). There is also an open [issue tracker](#) and various [tools for collecting bytecode statistics](#). CPU-intensive Python code in particular is likely to benefit from the changes; code already written in C, I/O-heavy processes and multithreaded code, on the other hand, are unlikely to benefit.

See also:

- [Faster CPython](#)

If you don't want to wait with your project until the release of Python 3.11 in the final version probably on 24 October 2022, you can also have a look at the following Python interpreters:

6.7.2 Cython

For intensive numerical operations, Python can be very slow, even if you have avoided all anti-patterns and used vectorisations with NumPy. In this case, translating code into [Cython](#) can be helpful. Unfortunately, the code often has to be restructured and thus increases in complexity. Explicit type annotations and the provision of code also become more cumbersome.

Our example could then look like this:

Listing 5: cy_kmeans.pyx

```
# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

cimport numpy as np
import numpy as np

cdef double dist(double[:] x, double[:] y):
    """Calculate the distance"""
    cdef double dist = 0
    for i in range(len(x)):
        dist += (x[i] - y[i]) ** 2
    return dist

def find_labels(double[:, :] points, double[:, :] centers):
    """Assign points to a cluster."""
    cdef int n_points = points.shape[0]
    cdef int n_centers = centers.shape[0]
    cdef double[:] labels = np.zeros(n_points)
    cdef double distance, nearest_distance
    cdef int nearest_index
```

(continues on next page)

(continued from previous page)

```

for i in range(n_points):
    nearest_distance = np.inf
    for j in range(n_centers):
        distance = dist(points[i], centers[j])
        if distance < nearest_distance:

```

See also:

- [Cython Tutorials](#)

6.7.3 Numba

Numba is a JIT compiler that translates mainly scientific Python and NumPy code into fast machine code, for example:

Listing 6: nb_kmeans.py

```

# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

import numba

@numba.jit(nopython=True)
def dist(x, y):
    """Calculate the distance"""
    dist = 0
    for i in range(len(x)):
        dist += (x[i] - y[i]) ** 2
    return dist

@numba.jit(nopython=True)
def find_labels(points, centers):
    """Assign points to a cluster."""
    labels = []
    min_dist = np.inf
    min_label = 0
    for i in range(len(points)):
        for j in range(len(centers)):
            distance = dist(points[i], centers[j])

```

However, Numba requires [LLVM](#) and some Python constructs are not supported.

6.8 Task planner

`ipyparallel`, `Dask` and `Ray` can distribute tasks in a cluster. In doing so, they have different focuses:

- `ipyparallel` simply integrates into a `JupyterHub`.
- `Dask` imitates `pandas`, `NumPy` iterators, `Toolz` und `PySpark` when it distributes their tasks.
- `Ray` provides a simple, universal API for building distributed applications.
 - `RLlib` will scale reinforcement learning in particular.
 - A backend for `joblib` supports distributed `scikit-learn` programs.
 - `XGBoost-Ray` is a backend for distributed `XGBoost`.
 - `LightGBM-Ray` is a backend for distributed `LightGBM`.
 - `Collective Communication Lib` offers a set of native collective primitives for `Gloo` and the `NVIDIA Collective Communication Library (NCCL)`.

Our example could look like this with `Dask`:

Listing 7: `ds_kmeans.py`

```
# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

import numpy as np
from dask import array as da
from dask import dataframe as dd

def find_labels(points, centers):
    """Assign points to a cluster."""
    diff = points[:, None, :] - centers
    distances = (diff**2).sum(-1)
    return distances.argmin(1)

def compute_centers(points, labels):
    """Calculate the cluster centres."""
    points_df = dd.from_dask_array(points)
    labels_df = dd.from_dask_array(labels)
    return points_df.groupby(labels_df).mean()

def kmeans(points, n_clusters):
    """Calculates the cluster centres repeatedly until nothing changes."""
    centers = points[-n_clusters:]
    points = da.from_array(points, 1000)
    while True:
        old_centers = centers
        labels = find_labels(points, da.from_array(centers, 5))
        centers = compute_centers(points, labels)
        centers = centers.compute().values
```

(continues on next page)

(continued from previous page)

```

    if np.all(centers == old_centers):
        break
    return labels.compute()

```

6.8.1 Dask

Dask performs two different tasks: 1. it optimizes dynamic task scheduling, similar to [Airflow](#), [Luigi](#) or [Celery](#). 2. it performs parallel data like arrays, dataframes, and lists with dynamic task scheduling.

Scales from laptops to clusters

Dask can be easily installed on a laptop with pipenv and expands the size of the datasets from *fits in memory* to *fits on disk*. Dask can also scale to a cluster of hundreds of machines. It is resilient, elastic, data-local and has low latency. For more information, see the [distributed scheduler](#) documentation. This simple transition between a single machine and a cluster allows users to both start easily and grow as needed.

Install Dask

You can install everything that is required for most common applications of Dask (arrays, dataframes, ...). This installs both Dask and dependencies such as NumPy, Pandas, etc. that are required for various workloads:

```
$ pipenv install "dask[complete]"
```

However, only individual subsets can be installed with:

```

$ pipenv install "dask[array]"
$ pipenv install "dask[dataframe]"
$ pipenv install "dask[diagnostics]"
$ pipenv install "dask[distributed]"

```

Testing the installation

```

[1]: !pytest /Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
    ↪ packages/dask/tests /Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/
    ↪ python3.11/site-packages/dask/array/tests

===== test session starts =====
platform darwin -- Python 3.11.4, pytest-7.4.0, pluggy-1.2.0
rootdir: /Users/veit
collected 5030 items / 18 skipped
../../../../../../../../local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/
    ↪ dask/tests/test_backends.py . [ 0%]
plugins: hypothesis-6.82.0, cov-4.1.0, anyio-3.7.1, typeguard-2.13.3
../../../../../../../../local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/
    ↪ dask/tests/test_base.py . [ 0%]
...
../../../../../../../../local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/
    ↪ dask/array/tests/test_xarray.py . [ 99%]
s [ 0%]

```

(continues on next page)

(continued from previous page)

```

..... [100%]

===== FAILURES =====
_____ test_solve_assume_a[20-10] _____
...
E       Failed: DID NOT RAISE <class 'FutureWarning'>

/Users/veit/.local/share/venv/python-311-6zxVKbDJ/lib/python3.11/site-packages/
↳dask/array/tests/test_linalg.py:809: Failed
_____ test_solve_assume_a[30-6] _____
...
E       Failed: DID NOT RAISE <class 'DeprecationWarning'>

/Users/veit/.local/share/venv/python-311-6zxVKbDJ/lib/python3.11/site-packages/
↳dask/array/tests/test_random.py:202: Failed
===== warnings summary =====
...
-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
===== short test summary info =====
FAILED ../../../../.local/share/venv/python-311-6zxVKbDJ/lib/python3.11/site-
↳packages/dask/array/tests/test_linalg.py::test_solve_assume_a[20-10] - Failed: DID NOT
↳RAISE <class 'FutureWarning'>
FAILED ../../../../.local/share/venv/python-311-6zxVKbDJ/lib/python3.11/site-
↳packages/dask/array/tests/test_linalg.py::test_solve_assume_a[30-6] - Failed: DID NOT
↳RAISE <class 'FutureWarning'>
FAILED ../../../../.local/share/venv/python-311-6zxVKbDJ/lib/python3.11/site-
↳packages/dask/array/tests/test_random.py::test_RandomState_only_funcs - Failed: DID
↳NOT RAISE <class 'DeprecationWarning'>
= 3 failed, 4543 passed, 487 skipped, 15 xfailed, 34 warnings in 60.13s (0:01:00) =

```

Familiar operation

Dask DataFrame

... imitates Pandas

```
[2]: import pandas as pd
```

```
df = pd.read_csv("2021-09-01.csv")
df.groupby(df.user_id).value.mean()
```

```
[3]: import dask.dataframe as dd
```

```
dd = pd.read_csv("2021-09-01.csv")
dd.groupby(dd.user_id).value.mean()
```

See also

- [Dask DataFrame Docs](#)

- [Dask DataFrame Best Practices](#)

Dask Array

... imitates NumPy

```
[4]: import numpy as np

f = h5py.File("mydata.h5")
x = np.array(f["."])
```

```
[5]: import dask.array as da

f = h5py.File("mydata.h5")
x = da.array(f["."])
```

See also

- [Dask Array Docs](#)
- [Dask Array Best Practices](#)

Dask Bag

... imitates iterators, Toolz und PySpark.

```
[6]: import json

import dask.bag as db

b = db.read_text("2021-09-01.csv").map(json.loads)
b.pluck("user_id").frequencies().topk(10, lambda pair: pair[1]).compute()
```

See also

- [Dask Bag Docs](#)

Dask Delayed

... imitates loops and wraps custom code

```
[7]: from dask import delayed

L = []
for fn in "2021-*-*.csv": # Use for loops to build up computation
    data = delayed(load)(fn) # Delay execution of function
    L.append(delayed(process)(data)) # Build connections between variables
```

(continues on next page)

(continued from previous page)

```
result = delayed(summarize)(L)
result.compute()
```

See also

- [Dask Delayed Docs](#)
- [Dask Delayed Best Practices](#)
- *[Dask pipeline example: Tracking the International Space Station with Dask](#)*

The `concurrent.futures` interface enables the submission of user-defined tasks.

Note

For the following example, Dask must be installed with the `distributed` option, e.g.

```
$ pipenv install dask[distributed]
```

```
[8]: from dask.distributed import Client

client = Client("scheduler:port")

futures = []
for fn in filenames:
    future = client.submit(load, fn)
    futures.append(future)

summary = client.submit(summarize, futures)
summary.result()
```

See also

- [Dask Futures Docs](#)
- [Dask Futures Quickstart](#)
- [Dask Futures Examples](#)

6.9 Multithreading, Multiprocessing and Async

After a brief *overview*, three examples of *threading*, *multiprocessing* and *async* illustrate the rules and best practices.

6.9.1 Introduction to multithreading, multiprocessing and async

Martelli's model of scalability

Number of cores	Description
1	Single thread and single process
2–8	Multiple threads and multiple processes
>8	Distributed processing

Martelli's observation was that over time the second category becomes less and less important as individual cores become more powerful and large data sets become larger.

Global Interpreter Lock (GIL)

CPython has a lock on its internally shared global state. As a result, no more than one thread can run at the same time.

The GIL is not a big problem for I/O-heavy applications; however, using threading will slow down CPU-heavy applications. Accordingly, multi-processing is exciting for us to get more CPU cycles.

[Literate programming](#) and *Martelli's model of scalability* determined the design decisions on Python's performance for a long time. Little has changed in this assessment to this day: Contrary to intuitive expectations, more CPUs and threads in Python initially lead to less efficient applications. However, the [Gilectomy](#) project, which was supposed to replace the GIL, also encountered another problem: the Python C API exposes too many implementation details. With this, however, performance improvements would quickly lead to incompatible changes, which then seem unacceptable, especially in a language as popular as Python.

Overview

Cri- te- rion	Multithreading	Multiprocessing	asyncio
Sep- a- ra- tion	Threads share one state. However, sharing a state can lead to race conditions, i.e. the result of an operation can depend on the timing of certain individual operations.	The processes are independent of each other. If they are to communicate with each other, interprocess communication (IPC) , object pickling and other overhead is necessary.	With <code>run_coroutine_threadsafe()</code> , <code>asyncio</code> objects can also be used by other threads. Almost all <code>asyncio</code> objects are not thread-safe.
Switc	Threads change preemptively , i.e. no explicit code needs to be added to cause a change of tasks. However, such a change is possible at any time; accordingly, critical areas must be protected with <code>lock</code> .	As soon as you get a process assigned, significant progress should be made. So you should not make too many roundtrips back and forth.	<code>asyncio</code> switches cooperatively , i.e. <code>yield</code> or <code>await</code> must be explicitly specified to cause a switch. You can therefore keep the effort to these changes very low.
Tool- ing	Threads require very little tooling: <code>Lock</code> and <code>Queue</code> . Locks are difficult to understand in non-trivial examples. For complex applications, it is therefore better to use atomic message queues or <code>asyncio</code> .	Simple tooling with <code>map</code> and <code>imap_unordered</code> among others, to test individual processes in a single thread before switching to multiprocessing. If IPC or object pickling is used, the tooling becomes more complex.	At least for complex systems, <code>asyncio</code> leads to the goal more easy than multithreading locks. However <code>asyncio</code> requires a large set of tools: <code>futures</code> , <code>Event Loops</code> and non-blocking versions of almost everything.
Per- for- manc	Multithreading produces the best results for IO-heavy tasks. The performance limit for threads is one CPU minus task switches and synchronisation overheads.	The processes can be distributed to several CPUs and should therefore be used for CPU-heavy tasks. However, additional effort may be required and synchronisation of the processes.	Calling a poor Python function takes more overhead than requesting a generator or awaitable – i.e., <code>asyncio</code> can utilise the CPU efficiently. For CPU-intensive tasks, however, multiprocessing is more suitable.

Summary

There is no one ideal implementation of concurrency – each of the approaches presented next has specific advantages and disadvantages. So before you decide which approach to follow, you should analyse the performance problems carefully and then choose the most suitable solution. In our projects, we often use several approaches, depending on the part for which the performance is to be optimised.

6.9.2 Threading example

Updating and displaying a counter:

```
[1]: counter = 0

print("Starting up")
for i in range(10):
    counter += 1
    print("The count is %d" % counter)
print("Finishing up")
```

```
Starting up
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
The count is 6
The count is 7
The count is 8
The count is 9
The count is 10
Finishing up
```

Start with code that is clear, simple, and top-down. It's easy to develop and incrementally testable.

Note

Test and debug your application before you start threading. Threading never makes debugging easier.

Convert to functions

The next step is to create reusable code as a function:

```
[2]: counter = 0

def worker():
    "My job is to increment the counter and print the current count"
    global counter

    counter += 1
    print("The count is %d" % counter)

print("Starting up")
for i in range(10):
    worker()
print("Finishing up")
```

```
Starting up
The count is 1
The count is 2
```

(continues on next page)

(continued from previous page)

```
The count is 3
The count is 4
The count is 5
The count is 6
The count is 7
The count is 8
The count is 9
The count is 10
Finishing up
```

Multi-Threading

Now some worker threads can be started:

```
[3]: import threading
```

```
counter = 0
```

```
def worker():
```

```
    "My job is to increment the counter and print the current count"
```

```
    global counter
```

```
    counter += 1
```

```
    print("The count is %d" % counter)
```

```
print("Starting up")
```

```
for i in range(10):
```

```
    threading.Thread(target=worker).start()
```

```
print("Finishing up")
```

```
Starting up
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
The count is 6
The count is 7
The count is 8
The count is 9
The count is 10
Finishing up
```

Test

A simple test run leads to the same result.

Detection of race conditions

Note

Tests cannot prove the absence of errors. Many interesting race conditions do not show up in test environments.

Fuzzing

Fuzzing is a technique to improve the detection of race conditions:

```
[4]: import random
import threading
import time

FUZZ = True

def fuzz():
    if FUZZ:
        time.sleep(random.random())

counter = 0

def worker():
    "My job is to increment the counter and print the current count"
    global counter

    fuzz()
    oldcnt = counter
    fuzz()
    counter = oldcnt + 1
    fuzz()
    print("The count is %d" % counter, end="")
    fuzz()

print("Starting up")
fuzz()
for i in range(10):
    threading.Thread(target=worker).start()
    fuzz()
print("Finishing up")
fuzz()
```

```

Starting up
The count is 1The count is 2The count is 2The count is 3The count is 3The count is 3The
↪count is 3Finishing up
The count is 4The count is 4The count is 5

```

This technique is limited to relatively small blocks of code and is imperfect in that it still cannot prove the absence of errors. Nevertheless, fuzzed tests can reveal race conditions.

Careful threading with queues

The following rules must be observed:

1. All shared resources should be executed in exactly one thread. All communication with this thread should be done with only one atomic message queue – usually with the `queue module`, email or message queues such as `RabbitMQ` or `ZeroMQ`.

Resources that require this technology include global variables, user inputs, output devices, files, sockets, etc.

2. One category of sequencing problems is to ensure that step A is performed before step B. The solution is to run them both on the same thread, with all the actions happening in sequence.
3. To implement a *barrier* that waits for all parallel threads to complete, just join all threads with `join()`.
4. You cannot wait for daemon threads to complete (they are infinite loops); instead you should execute `join()` on the queue itself, so that the tasks are only merged when all tasks in the queue have been completed.
5. You can use global variables to communicate between functions, but only within a single-threaded program. In a multi-thread program, however, you cannot use global variables because they are mutable. Then the better solution is `threading.local()`, since it is global in a thread, but not beyond.
6. Never try to terminate a thread from the outside: you never know if that thread is holding a lock. Therefore, Python does not provide a direct thread termination mechanism. However, if you try to do this with `ctypes`, this is a recipe for deadlock.

Now, if we apply these rules, our code looks like this:

```

[5]: import queue
import threading

counter = 0

counter_queue = queue.Queue()

def counter_manager():
    "I have EXCLUSIVE rights to update the counter variable"
    global counter

    while True:
        increment = counter_queue.get()
        counter += increment
        print_queue.put(
            [
                "The count is %d" % counter,
            ]

```

(continues on next page)

(continued from previous page)

```

    )
    counter_queue.task_done()

t = threading.Thread(target=counter_manager)
t.daemon = True
t.start()
del t

print_queue = queue.Queue()

def print_manager():
    while True:
        job = print_queue.get()
        for line in job:
            print(line)
        print_queue.task_done()

t = threading.Thread(target=print_manager)
t.daemon = True
t.start()
del t

def worker():
    "My job is to increment the counter and print the current count"
    counter_queue.put(1)

print_queue.put(["Starting up"])
worker_threads = []
for i in range(10):
    t = threading.Thread(target=worker)
    worker_threads.append(t)
    t.start()
for t in worker_threads:
    t.join()

counter_queue.join()
print_queue.put(["Finishing up"])
print_queue.join()

```

```

Starting up
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
The count is 6
The count is 7

```

(continues on next page)

(continued from previous page)

```
The count is 8
The count is 9
The count is 10
Finishing up
```

Careful threading with locks

If we thread with locks instead of queues, the code looks even tidier:

```
[6]: import random
import threading
import time

counter_lock = threading.Lock()
printer_lock = threading.Lock()

counter = 0

def worker():
    global counter
    with counter_lock:
        counter += 1
    with printer_lock:
        print("The count is %d" % counter)

with printer_lock:
    print("Starting up")

worker_threads = []
for i in range(10):
    t = threading.Thread(target=worker)
    worker_threads.append(t)
    t.start()
for t in worker_threads:
    t.join()

with printer_lock:
    print("Finishing up")
```

```
Starting up
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
The count is 6
The count is 7
The count is 8
```

(continues on next page)

(continued from previous page)

```
The count is 9
The count is 10
Finishing up
```

Finally, a few notes on locks:

1. Locks are just so-called *flags*, they are not really reliable.
2. In general, locks should be viewed as a primitive tool that is difficult to understand in non-trivial examples. For more complex applications, it is better to use atomic message queues.
3. The more locks that are set at the same time, the less the benefits of simultaneous processing.

6.9.3 Multi-processing example

We'll start with code that is clear, simple, and executed top-down. It's easy to develop and incrementally testable:

```
[1]: from multiprocessing.pool import ThreadPool as Pool

import requests

sites = [
    "https://github.com/veit/jupyter-tutorial/",
    "https://jupyter-tutorial.readthedocs.io/en/latest/",
    "https://github.com/veit/pyviz-tutorial/",
    "https://pyviz-tutorial.readthedocs.io/de/latest/",
    "https://cusy.io/en",
]

def sitesize(url):
    with requests.get(url) as u:
        return url, len(u.content)

pool = Pool(10)
for result in pool.imap_unordered(sitesize, sites):
    print(result)

('https://cusy.io/en', 36389)
('https://jupyter-tutorial.readthedocs.io/en/latest/', 40884)
('https://github.com/veit/jupyter-tutorial/', 236862)
('https://github.com/veit/pyviz-tutorial/', 213124)
('https://pyviz-tutorial.readthedocs.io/de/latest/', 32803)
```

Note

A good development strategy is to use `map`, to test your code in a single process and thread before moving to multi-processing.

Note

In order to better assess when `ThreadPool` and when `process Pool` should be used, here are some rules of thumb:

- For CPU-heavy jobs, `multiprocessing.pool.Pool` should be used. Usually we start here with twice the number of CPU cores for the pool size, but at least 4.
- For I/O-heavy jobs, `multiprocessing.pool.ThreadPool` should be used. Usually we start here with five times the number of CPU cores for the pool size.
- If we use Python 3 and do not need an interface identical to `pool`, we use `concurrent.future.Executor` instead of `multiprocessing.pool.ThreadPool`; it has a simpler interface and was designed for threads from the start. Since it returns instances of `concurrent.futures.Future`, it is compatible with many other libraries, including `asyncio`.
- For CPU- and I/O-heavy jobs, we prefer `multiprocessing.Pool` because it provides better process isolation.

```
[2]: from multiprocessing.pool import ThreadPool as Pool

import requests

sites = [
    "https://github.com/veit/jupyter-tutorial/",
    "https://jupyter-tutorial.readthedocs.io/en/latest/",
    "https://github.com/veit/pyviz-tutorial/",
    "https://pyviz-tutorial.readthedocs.io/de/latest/",
    "https://cussy.io/en",
]

def sitesize(url):
    with requests.get(url) as u:
        return url, len(u.content)

for result in map(sitesize, sites):
    print(result)

('https://github.com/veit/jupyter-tutorial/', 236862)
('https://jupyter-tutorial.readthedocs.io/en/latest/', 40884)
('https://github.com/veit/pyviz-tutorial/', 213124)
('https://pyviz-tutorial.readthedocs.io/de/latest/', 32803)
('https://cussy.io/en', 36389)
```

What can be parallelised?

Amdahl's law

The increase in speed is mainly limited by the sequential part of the problem, since its execution time cannot be reduced by parallelisation. In addition, parallelisation creates additional costs, such as for communication and synchronisation of the processes.

In our example, the following tasks can only be processed serially:

- UDP DNS request request for the URL
- UDP DNS response
- Socket from the OS

- TCP-Connection
- Sending the HTTP request for the root resource
- Waiting for the TCP response
- Counting characters on the site

```
[3]: from multiprocessing.pool import ThreadPool as Pool

import requests

sites = [
    "https://github.com/veit/jupyter-tutorial/",
    "https://jupyter-tutorial.readthedocs.io/en/latest/",
    "https://github.com/veit/pyviz-tutorial/",
    "https://pyviz-tutorial.readthedocs.io/de/latest/",
    "https://cusy.io/en",
]

def sitesize(url):
    with requests.get(url, stream=True) as u:
        return url, len(u.content)

pool = Pool(4)
for result in pool.imap_unordered(sitesize, sites):
    print(result)

('https://github.com/veit/jupyter-tutorial/', 236862)
('https://github.com/veit/pyviz-tutorial/', 213124)
('https://pyviz-tutorial.readthedocs.io/de/latest/', 32803)
('https://jupyter-tutorial.readthedocs.io/en/latest/', 40884)
('https://cusy.io/en', 36389)
```

Note

`imap_unordered` is used to improve responsiveness. However, this is only possible because the function returns the argument and result as a tuple.

Tips

- Don't make too many trips back and forth

If you get too many iterable results, this is a good indicator of too many trips, such as in

```
>>> def sitesize(url, start):
...     req = urllib.request.Request()
...     req.add_header('Range:%d-%d' % (start, start+1000))
...     u = urllib.request.urlopen(url, req)
...     block = u.read()
...     return url, len(block)
```

- Make relevant progress on every trip

Once you get the process, you should make significant progress and not get bogged down. The following example illustrates intermediate steps that are too small:

```
>>> def sitesize(url, results):
...     with requests.get(url, stream=True) as u:
...         while True:
...             line = u.iter_lines()
...             results.put((url, len(line)))
```

- Don't send or receive too much data

The following example unnecessarily increases the amount of data:

```
>>> def sitesize(url):
...     with requests.get(url) as u:
...         return url, u.content
```

6.9.4 Threading and forking combined

Mixing multiprocessing and threading is generally problematic and a recipe for deadlocks.

The following code was entered in 2016 at <https://bugs.python.org/issue27422> in the Python bug tracker:

```
[1]: import multiprocessing
import subprocess
import sys

from concurrent.futures import ThreadPoolExecutor

def run(arg):
    print("starting %s" % arg)
    p = multiprocessing.Process(target=print, args=("running", arg))
    p.start()
    p.join()
    print("finished %s" % arg)

if __name__ == "__main__":
    n = 16
    tests = range(n)
    with ThreadPoolExecutor(n) as pool:
        for r in pool.map(run, tests):
            pass

starting 0starting 1

starting 2
starting 3
starting 4
starting 5
starting 6
starting 7
starting 8
```

(continues on next page)

(continued from previous page)

```
starting 9
starting 10
starting 11
starting 12
starting 13
starting 14
starting 15
running 0
finished 4
running 4
running 1
running 2
running 3
running 15
finished 15
finished 0
finished 3
finished 1
finished 2
running 5
finished 5
running 7
running 6
finished 7
running 11
finished 6
running 9
finished 11
finished 9
running 8
running 10
finished 8
finished 10
running 13
finished 13
running 12
running 14
finished 12
finished 14
```

Usually, threading is recommended after the fork, not before. Otherwise, the locks used when executing the threads are duplicated across the processes. If one of these processes dies with a lock, all other processes with this lock are deadlocked.

6.9.5 asyncio example

From IPython7.0 you can use asyncio directly in Jupyter Notebooks, see also [IPython 7.0, Async REPL](#).

If you get `RuntimeError: This event loop is already running`, `[nest-asyncio]` might help you.

Ihr könnt das Paket installieren mit

```
$ pipenv install nest-asyncio
```

You can then import it into your notebook and use it with:

```
[1]: import nest_asyncio
```

```
nest_asyncio.apply()
```

See also

- [asyncio: We Did It Wrong](#) by Lynn Root
- [An Intro to asyncio](#) by Mike Driscoll
- [Asyncio Coroutine Patterns: Beyond await](#) by Yeray Diaz

Simple *Hello world* example

```
[2]: import asyncio
```

```
async def hello():  
    print("Hello")  
    await asyncio.sleep(1)  
    print("world")
```

```
await hello()
```

```
Hello  
world
```

A little bit closer to a real world example

```
[3]: import asyncio  
import random
```

```
async def produce(queue, n):  
    for x in range(1, n + 1):  
        # produce an item  
        print("producing {}/{}".format(x, n))  
        # simulate i/o operation using sleep  
        await asyncio.sleep(random.random())  
        item = str(x)
```

(continues on next page)

(continued from previous page)

```

    # put the item in the queue
    await queue.put(item)

    # indicate the producer is done
    await queue.put(None)

async def consume(queue):
    while True:
        # wait for an item from the producer
        item = await queue.get()
        if item is None:
            # the producer emits None to indicate that it is done
            break

        # process the item
        print("consuming {}".format(item))
        # simulate i/o operation using sleep
        await asyncio.sleep(random.random())

loop = asyncio.get_event_loop()
queue = asyncio.Queue()
asyncio.ensure_future(produce(queue, 10), loop=loop)
loop.run_until_complete(consume(queue))

```

```

producing 1/10
producing 2/10
consuming 1
producing 3/10
consuming 2
producing 4/10
consuming 3
producing 5/10
consuming 4
producing 6/10
consuming 5
producing 7/10
consuming 6
producing 8/10
consuming 7
producing 9/10
consuming 8
producing 10/10
consuming 9
consuming 10

```

Exception Handling

See also

- `set_exception_handler`

```
[4]: def main():
    loop = asyncio.get_event_loop()
    # May want to catch other signals too
    signals = (signal.SIGHUP, signal.SIGTERM, signal.SIGINT)
    for s in signals:
        loop.add_signal_handler(
            s, lambda s=s: asyncio.create_task(shutdown(loop, signal=s))
        )
    loop.set_exception_handler(handle_exception)
    queue = asyncio.Queue()
```

Testing with pytest

Example:

```
[5]: import pytest

@pytest.mark.asyncio
async def test_consume(mock_get, mock_queue, message, create_mock_coro):
    mock_get.side_effect = [message, Exception("break while loop")]

    with pytest.raises(Exception, match="break while loop"):
        await consume(mock_queue)
```

Third-party libraries

- `pytest-asyncio` has helpful things like fixtures for `event_loop`, `unused_tcp_port`, and `unused_tcp_port_factory`; and the ability to create your own `asynchronous fixtures`.
- `asynctest` has helpful tooling, including coroutine mocks and `exhaust_callbacks` so we don't have to manually await tasks.
- `aiohttp` has some really nice built-in test utilities.

Debugging

`asyncio` already has a `debug mode` in the standard library. You can simply activate it with the `PYTHONASYNCIODEBUG` environment variable or in the code with `loop.set_debug(True)`.

Using the debug mode to identify slow async calls

asyncio's debug mode has a tiny built-in profiler. When debug mode is on, asyncio will log any asynchronous calls that take longer than 100 milliseconds.

Debugging in production with aiodebug

aiodebug is a tiny library for monitoring and testing asyncio programs.

Example

```
[6]: from aiodebug import log_slow_callbacks

def main():
    loop = asyncio.get_event_loop()
    log_slow_callbacks.enable(0.05)
```

Logging

aiologger allows non-blocking logging.

Asynchronous Widgets

See also

- [Asynchronous Widgets](#)

```
[7]: def wait_for_change(widget, value):
    future = asyncio.Future()

    def getvalue(change):
        # make the new value available
        future.set_result(change.new)
        widget.unobserve(getvalue, value)

    widget.observe(getvalue, value)
    return future
```

```
[8]: from ipywidgets import IntSlider

slider = IntSlider()

async def f():
    for i in range(10):
        print("did work %s" % i)
        x = await wait_for_change(slider, "value")
```

(continues on next page)

(continued from previous page)

```
print("async function continued with value %s" % x)
```

```
asyncio.ensure_future(f())
```

```
slider
```

```
IntSlider(value=0)
```

```
did work 0
```

CREATE A PRODUCT

With Jupyter Notebooks you can quickly build prototypes for data analysis. You can also use them to document and present your results. However, they are not well suited for reproducing your results with *Kernel → Restart and Run All* in a few days or years. For example, the notebooks contain very little information about the environment, the `kernel`, with which they have been able to run successfully in the past. Although you can use `pd.show_versions()` to display *Information about the host operating system and the versions of installed Python packages*, this is unfortunately not sufficient to reproduce such an environment.

«Non-reproducible single occurrences are of no significance to science.»¹

In order for others to be able to use your code, it should meet some conditions:

- You should not silently rely on specific resources and environments
- Required software packages and hardware should be specified in the requirements
- Path information will only work in a different context within your package or in previously generated directories and files
- Do not share secrets like login details or internal IP numbers in your published product

There are various tools that support you in creating shareable products. These can be tools on the one hand for the *versioning of the source code* and the *training data* as well as for the reproducibility of the *execution environments*, on the other hand for *Testing, Logging, documenting* and *creating packages*.

See also:

- Dustin Boswell, Trevor Foucher: *The Art of Readable Code*
- TIB workshop «FAIR Data and Software»
 - [GitHub Page](#)
 - [GitHub Repository](#)
 - [Slides](#)
- Dryad: [Best practices for creating reusable data publications](#)

¹ Karl Popper in *The Logic of Scientific Discovery*, 1959

7.1 Manage code with Git

To gain better control over your source code, it is usually managed with [Git](#). [Git](#) is a mature and very actively maintained open source project originally developed in 2005 by Linus Torvalds, the initiator of the Linux operating system kernel. [Git](#) can be combined well with many operating systems and IDEs (integrated development environments).

With its distributed architecture, [Git](#) is an example of a DVCS (distributed version control system). This means that the entire version history no longer has to be in a single location, as was common with previously popular version control systems such as CVS or Subversion (SVN). In [Git](#), each local repository can contain specific changes.

However, [Git](#) can not only be used in a distributed way, it is also performant, secure and flexible.

7.1.1 Performance

[Git](#) is very fast compared to many other version control systems in committing changes, branching and merging, and comparing with previous versions. This is also necessary when we look at the [Linux kernel repository](#) with over a million commits. [Git](#) is not oriented towards file names, but focuses on changes in content so that files can be efficiently renamed, split and rearranged. [Git](#) achieves this by storing deltas for the differences in content, metadata of the files and compression.

The distributed version control system also ensures that, for example, implementing a new function does not require network access to a remote server, thus avoiding delays. You can also carry out error correction locally on an earlier version. Later, both changes can be transmitted to a central server with a single command.

7.1.2 Security

The integrity of managed source code was a high priority in the design of [Git](#). For example, the relationships between files and commits are protected by a hashing algorithm (SHA1), making accidental or deliberate changes more difficult and ensuring the actual history.

7.1.3 Flexibility

[Git](#) not only allows for very flexible workflows but is also suitable for both large and small projects on different platforms.

7.1.4 Criticisms

A common criticism of [Git](#) is that it is difficult to learn: either large parts of the [Git](#) terminology are new or in other systems terms have a different meaning, such as for example `revert` in SVN or CVS. [Git](#) also offers a lot of functionality, but it takes some time to learn.



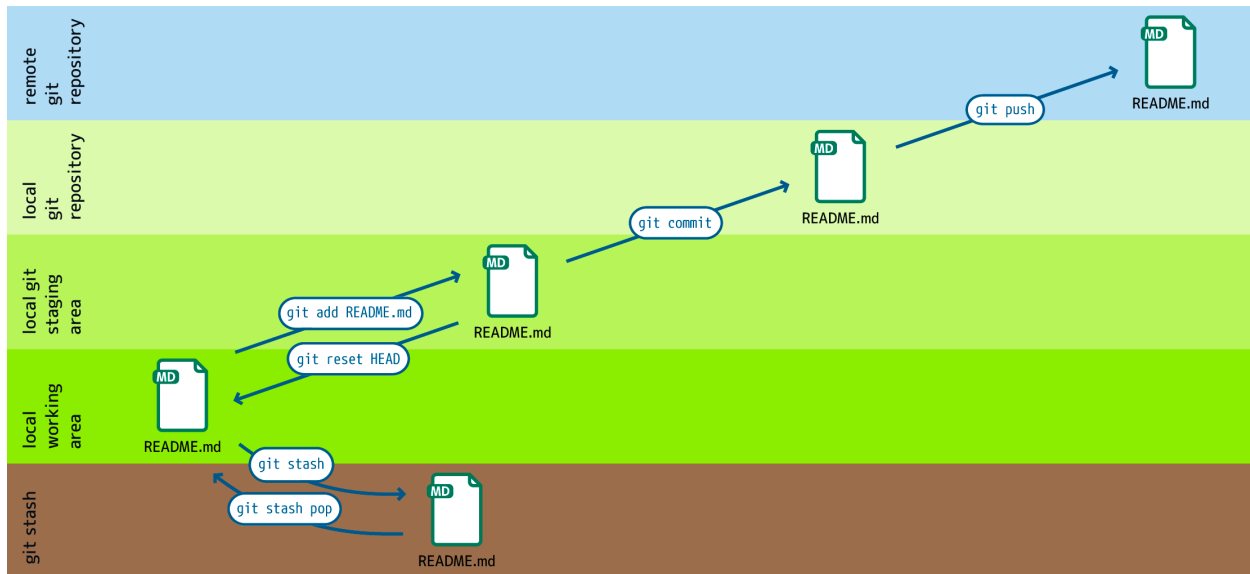
7.1.5 Read more

See also:

- [Git Cheat Sheet \(PDF\)](#)
- [Interactive Git Cheatsheet](#)
- [Software Carpentry Version Control with Git](#)
- [Flight rules for Git](#)
- [First Aid git](#)
- [git-tips](#)
- [Pro Git book](#)
- [Git reference](#)

Essentially, in this tutorial, I show on the one hand how [Jupyter Notebooks](#) can be managed with Git, and on the other hand *best practices* and typical *Git workflows*.

Workspaces



Git manages multiple locations or **workspaces** where files are stored:

local working copy

contains files and folders that can be edited normally.

staging area

contains changes to files that are scheduled for writing into the version history.

local repository

contains the entire history of all files in the project.

remote repository

also contains the entire history, but is stored on a remote server.

stash

contains changes that are temporarily stored somewhere else to move them out of the way.

Basic Git commands

The following basic Git commands move changes between these workspaces.

git add

adds files from the working directory to the staging area.

git reset HEAD

restores a file in the work area from the stage area.

git stash

moves files from the workspace to a stash.

git stash pop

brings files from the stash to the work area.

git commit

writes changes from the staging area to the local repository.

git pull

copies changes from the remote to the local repository and updates the work area.

git push

copies changes from the local repository to the remote repository.

git push -u UPSTREAM BRANCHNAME

-u (long form --set-upstream)

allows to specify the remote repository and a branch in it.

UPSTREAM

the name of the remote repository, typically `origin`.

BRANCHNAME

the name of a branch in the remote repository, typically the same as in the local repository.

Git installation and configuration

Installation

For iX distributions, Git should be in the standard repository.

The `git-all` package provides a complete Git working environment. Install it with:

```
$ sudo apt install git-all
```

To install only Git the `git` package suffices:

```
$ sudo apt install git
```

The bash autocompletion makes Git easier to use on the command line. The according package is called `bash-completion`. Install it with:

```
$ sudo apt install bash-completion
```

There are several different ways to install Git on a Mac. Probably the easiest way to do is to install the Xcode Command Line Tools. For this you only have to call up `git` in the terminal for the first time:

```
$ git --version
```

`git-completion` you can install with `Homebrew`:

Then you have to add the following line to the file `~/.bash_profile`:

```
[[ -r "$(brew --prefix)/etc/profile.d/bash_completion.sh" ]] && . "$(brew --prefix)/etc/
↪profile.d/bash_completion.sh"
```

Go to <https://git-scm.com/download/win> and start the download automatically. Further information can be found at <https://gitforwindows.org/>.

Configuration

The author of every change needs to be transparent. Specify your name and email address as follows:

```
$ git config --global user.name "NAME"
```

defines the name *NAME* associated with your commit transactions.

```
$ git config --global user.email "EMAIL-ADDRESS"
```

defines the email address *EMAIL-ADDRESS* that will be linked to your commit transactions.

For better readability, activate the coloring of the command line output:

```
$ git config --global color.ui auto
```

The ~/.gitconfig file

For example, the following file can be created with the commands given above:

```
[user]
  name = veit
  email = veit@cusy.io

[color]
  ui = auto
```

However, aliases can also be specified in the ~/.gitconfig file:

```
[alias]
  st = status
  ci = commit
  br = branch
  co = checkout
  df = diff
  dfs = diff --staged
```

See also:

Shell-Konfiguration:

- [oh-my-zsh](#)
 - [Git plugin aliases](#)
 - [zsh-you-should-use](#)
- [Starship](#)
 - [git_branch-Modul](#)
 - [git_commit-Modul](#)
 - [git_state](#)
 - [git_status-Modul](#)

The editor can also be specified, for example with:

```
[core]
  editor = vim
```

or for Visual Studio Code:

```
[core]
  editor = code --wait
```

Note: On macOS, you must first start Visual Studio Code, then open the command palette with `⌘+P` and finally execute the `Install 'code' command in PATH`.

The highlighting of space errors in `git diff` can also be configured:

```
[core]
# Highlight whitespace errors in git diff:
whitespace = tabwidth=4,tab-in-indent,cr-at-eol,trailing-space
```

Note: In addition to `~/.gitconfig`, since version 1.17.12 Git also looks in `~/.config/git/config` for a global configuration file.

Under Linux, `~/.config` can sometimes be a different path set by the environment variable `XDG_CONFIG_HOME`. This behaviour is part of the [X Desktop Group \(XDG\) specification](#). You can get the other path with:

```
$ echo $XDG_CONFIG_HOME
```

See also:

- [git config files](#)

Since you can set options at multiple levels, you may want to keep track of where Git reads a particular value from. With `git config --list`¹ you can list all the overridden options and values. You can combine this with `--show-scope`² to see where Git is getting the value from:

```
$ git config --list --show-scope
system credential.helper=osxkeychain
global user.name=veit
global user.email=veit@cusy.io
...
```

You can also use `--show-origin`³ to list the names of the configuration files:

```
$ git config --list --show-origin
file:/opt/homebrew/etc/gitconfig credential.helper=osxkeychain
file:/Users/veit/.config/git/config user.name=veit
file:/Users/veit/.config/git/config user.email=veit@cusy.io
...
```

¹ `git config --list`

² `git config --show-scope`

³ `git config --show-origin`

Alternative configuration file

You can use other configuration files for certain working directories, for example to distinguish between private and professional projects. You can use a local configuration in your repository or [conditional includes](#) at the end of your global configuration:

```
...
[includeIf "gitdir:~/private"]
path = ~/.config/git/config-private
```

This construct ensures that Git includes additional configurations or overwrites existing ones when you work in `~/private`.

Now create the file `~/.config/git/config-private` and define your alternative configuration there, for example:

```
[user]
  email = kontakt@veit-schiele.de
[core]
  sshCommand = ssh -i ~/.ssh/private_id_rsa
```

See also:

- `core.sshCommand`

Manage login data

Since Git version 1.7.9, the access data to git repositories can be managed with [gitcredentials](#). To use this, you can, for example, specify the following:

```
$ git config --global credential.helper Cache
```

This will keep your password in the cache for 15 minutes. If necessary, the timeout can be increased, for example with:

```
$ git config --global credential.helper 'cache --timeout=3600'
```

With macOS you can use *osxkeychain* to store the login information. *osxkeychain* requires Git version 1.7.10 or newer and can be installed in the same directory as Git with:

```
$ git credential-osxkeychain
git: 'credential-osxkeychain' is not a git command. See 'git --help'.
$ curl -s -O http://github-media-downloads.s3.amazonaws.com/osx/git-credential-
→osxkeychain
$ chmod u+x git-credential-osxkeychain
$ sudo mv git-credential-osxkeychain /usr/bin/
Password:
git config --global credential.helper osxkeychain
```

This enters the following in the `~/.gitconfig` file:

```
[credential]
  helper = osxkeychain
```

For Windows, [Git Credential Manager \(GCM\)](#) is available. It is integrated in [Git for Windows](#) and is installed by default. However, there is also a standalone Installer in [Releases](#).

It is configured with

```
$ git credential-manager configure
Configuring component 'Git Credential Manager'...
Configuring component 'Azure Repos provider'...
```

This will add the [credential] section to your ~/.gitconfig file:

```
[credential]
  helper =
  helper = C:/Program\ Files/Git/mingw64/bin/git-credential-manager.exe
```

Now, when cloning a repository, a *Git Credential Manager* window opens and asks you to enter your credentials.

In addition, the ~/.gitconfig file is supplemented, for example by the following two lines:

```
[credential "https://ce.cusy.io"]
  provider = generic
```

Note: You can find a comprehensive example of a ~/.gitconfig file in my [dotfiles](#) repository: [.gitconfig](#).

See also:

- [Git Credential Manager: authentication for everyone](#)

The .gitignore file

In the .gitignore file you can exclude files from version management. A typical .gitignore file can look like this:

```
/logs/*
!logs/.gitkeep
/tmp
*.swp
```

In doing so, Git uses [Globbing](#) patterns, among others:

Pattern	Example	Description
<code>**/logs</code>	logs/instance.log, logs/ instance/error.log, prod/ logs/instance.log	You can put two asterisks to prefix directories anywhere.
<code>**/logs/instance.log</code>	logs/instance.log, prod/ logs/instance.log but not logs/prod/instance.log	You can put two asterisks to prefix files with their name in a parent directory.
<code>*.log</code>	instance.log, error.log, logs/instance.log	An asterisk is a placeholder for null or more characters.
<code>/logs</code> <code>!/logs/.gitkeep</code>	/logs/instance.log, /logs/ error.log, but not /logs/. gitkeep or /instance.log	An exclamation mark in front of a pattern ignores it. If a file matches a pattern, but also a negating one that is defined later, it is not ignored.
<code>/instance.log</code>	/instance.log, but not logs/ instance.log	With a preceding slash, the pattern only matches files in the root directory of the repository.
<code>instance.log</code>	instance.log, logs/instance. log	Usually the pattern match files in any directory.
<code>instance?.log</code>	instance0.log, instance1. log, but not instance.log or instance10.log	A question mark fits exactly on a character.
<code>instance[0-9].log</code>	instance0.log, instance1. log, but not instance.log or instance10.log	Square brackets can be used to find a single character from a specific range.
<code>instance[01].log</code>	instance0.log, instance1. log, but not instance2.log or instance01.log	Square brackets match a single character from a given set.
<code>instance[!01].log</code>	instance2.log, but not instance0.log, instance1. log or instance01.log	An exclamation mark can be used to find any character from a specified set.
<code>logs</code>	logs logs/instance.log prod/ logs/instance.log	If no slash appended, the pattern fix both files and the contents of directories with this name.
<code>logs/</code>	logs/instance.log, logs/ prod/instance.log, prod/ logs/instance.log	Appending a slash indicates that the pattern is a directory. The entire contents of any directory in the repository that matches the name – including all its files and subdirectories – are ignored.
<code>var/**/instance.log</code>	var/instance.log, var/logs/ instance.log, but not var/logs/ instance/error.log	Two Asterisks match null or more directories.
<code>logs/instance*/error.log</code>	logs/instance/error.log, logs/instance1/error.log	Wildcards can also be used in directory names.
<code>logs/instance.log</code>	logs/instance.log, but not var/logs/instance.log or instance.log	Pattern, that specify a particular file in a directory are relative to the root of the repository.

Git-commit empty folder

In the example above you can see that with `/logs/*` no content of the `logs` directory should be versioned with Git, but an exception is defined in the following line: `!logs/.gitkeep` allows the file `.gitkeep` to be managed with Git. The `logs` directory is then also transferred to the Git repository. This construction is necessary because empty folders cannot be managed with Git.

Another possibility is to create a `.gitignore` file in an empty folder with the following content:

```
# ignore everything except .gitignore
*
!.gitignore
```

excludesfile

However, you can also exclude files centrally for all Git repositories. For this purpose, you can set `excludesfile` in the `~/.gitconfig` file:

```
[core]

# Use custom `.gitignore`
excludesfile = ~/.gitignore
...
```

Note: You can find helpful templates in my [dotfiles](#) repository or on the [gitignore.io](#) website.

Ignoring a file from the repository

If you want to ignore a file that has already been added to the repository in the past, you need to delete the file from your repository and then add a `.gitignore` rule for it. Using the `--cached` option on `git rm` means that the file will be deleted from the repository but will remain in your working directory as an ignored file.

```
$ echo *.log >> .gitignore
$ git rm --cached *.log
rm 'instance.log'
$ git commit -m "Remove log files"
```

Note: You can omit the `--cached` option if you want to remove the file from both the repository and your local file system.

Commit an ignored file

It is possible to force the commit of an ignored file to the repository with the `-f` (or `--force`) option on `git add`:

```
$ cat data/.gitignore
*
$ git add -f data/iris.csv
$ git commit -m "Force add iris.csv"
```

You might consider this if you have a general pattern (like `*`) defined, but want to commit a specific file. However, a better solution is usually to define an exception to the general rule:

```
$ echo '!iris.csv' >> data/.gitignore
$ cat data/.gitignore
*
!iris.csv
$ git add data/iris.csv
$ git commit -m "Add iris.csv"
```

This approach should be more obvious and less confusing for your team.

Troubleshooting .gitignore files

For complicated `.gitignore` patterns, or patterns that are spread across multiple `.gitignore` files, it can be difficult to figure out why a particular file is being ignored.

With `git status --ignored=matching`⁴, an *Ignored Files* section is added to the output, showing all ignored files and directories:

```
$ git status --ignored=matching
On branch main
Ignored Files:
  (use "git add -f <file>...", to pre-mark the changes for committing)
  .DS_Store
  docs/.DS_Store
  docs/_build/doctrees/
  docs/_build/html/
  docs/clean-prep/.ipynb_checkpoints/
  ...
nothing to commit, working tree clean
```

You can use the `git check-ignore` command with the `-v` (or `--verbose`) option to determine which pattern is causing a particular file to be ignored:

```
$ git check-ignore -v data/iris.csv
data/.gitignore:2:!iris.csv  data/iris.csv
```

The output shows `FILE_CONTAINING_THE_PATTERN:LINE_NUMBER_OF_THE_PATTERN:PATTERN FILE_NAME`

You can pass multiple filenames to `git check-ignore` if you like, and the names themselves don't even have to match the files that exist in your repository.

⁴ `git status --ignored`

You can get a complete list of all ignored files with `git ls-files --ignored --exclude-standard --others`⁵. With `--exclude-standard` the standard ignored files are read and with `--others` the non-versioned files are displayed instead of the versioned ones:

```
$ git ls-files --ignored --exclude-standard --others
.DS_Store
_build/doctrees/clean-prep/bulwark.doctree
_build/doctrees/clean-prep/dask-pipeline.doctree
_build/doctrees/clean-prep/deduplicate.doctree
...
```

Occasionally you may want to bypass the global `~/.gitignore` file to see which files Git always ignores, regardless of your configuration. You can do this by switching to another exclude option, `--exclude-per-directory`, which uses only the repository's `.gitignore` files:

```
$ git ls-files --ignored --exclude-per-directory=.gitignore --others
docs/_build/doctrees/clean-prep/bulwark.doctree
docs/_build/doctrees/clean-prep/dask-pipeline.doctree
docs/_build/doctrees/clean-prep/deduplicate.doctree
...
```

Note that the `.DS_Store` file is no longer listed as ignored.

If you replace `--others` with `--cached`, `git ls-files` will list files that would be ignored unless they have already been committed:

```
$ git ls-files --ignored --exclude-per-directory=.gitignore --cached
data/iris.csv
```

You may have such files because someone added them to a `.gitignore` file before the relevant patterns, or because someone added them with `git add --force`. Either way, if you no longer want to manage the file with Git, you can remove it from Git management with the following one-liner, but don't delete it:

```
$ git ls-files --ignored --exclude-per-directory=.gitignore --cached | xargs -r git rm --
↪cached
rm 'data/iris.csv'
```

Working with Git

Start working on a project

Start your own project

```
$ git init [PROJECT]
```

creates a new, local git repository.

[PROJECT]

if the project name is given, Git creates a new directory and initializes it.

If no project name is given, the current directory is initialised.

⁵ `git check-ignore`

Work on a project

\$ **git clone** *PROJECT_URL*

downloads a project with all branches and the entire history from the remote repository.

--depth

indicates the number of commits to be downloaded.

-b

specifies the name of the remote branch to be downloaded.

Work on a project

\$ **git status**

shows the status of the current branch in the working directory with new, changed and files already marked for commit.

-v

shows the changes in the stage area as a diff.

-vv

also shows the changes in the working directory as a second diff.

See also:

[git status -v](#)

\$ **git add** *PATH*

adds one or more files to the stage area.

-p

adds parts of one or more files to the stage area.

-e

the changes to be adopted can be edited in the standard editor.

\$ **git diff** [*PATH*]

shows differences between working and stage areas, for example:

```
$ git diff docs/productive/git/work.rst
diff --git a/docs/productive/git/work.rst b/docs/productive/git/work.rst
index e2a5ea6..fd84434 100644
--- a/docs/productive/git/work.rst
+++ b/docs/productive/git/work.rst
@@ -46,7 +46,7 @@

:samp:`$ git diff {FILE}`
-   shows differences between work and stage areas.
+   shows differences between work and stage areas, for example:
```

index e2a5ea6..fd84434 100644 displays some internal Git metadata that you will probably never need. The numbers correspond to the hash identifiers of the git object versions.

The rest of the output is a list of diff chunks whose header is enclosed in @@ symbols. Each chunk shows changes made in a file. In our example, 7 lines were extracted starting at line 46 and 7 lines were added starting at line 46.

By default, `git diff` performs the comparison against `HEAD`. If you use `git diff HEAD docs/productive/git/work.rst` in the example above, it will have the same effect.

`git diff` can be passed Git references. Besides `HEAD`, some other examples of references are tags and branch names, for example `git diff MAIN..FEATURE_BRANCH`. The dot operator in this example indicates that the diff input is the tips of the two branches. The same effect occurs if the dots are omitted and a space is used between the branches. In addition, there is a three-dot operator: `git diff MAIN...FEATURE_BRANCH`, which initiates a diff where the first input parameter `MAIN` is changed so that the reference is the common ancestor of `MAIN` and `FEATURE`.

Every commit in Git has a commit ID, which you can get by running `git log`. You can then also pass this commit ID to `git diff`:

```
$ git log --pretty=oneline
af1a395a08221ffa83b46f562b6823cf044a108c (HEAD -> main, origin/main, origin/HEAD) :
↪memo: Add some git diff examples
d650de52306b63b93e92bba4f15be95eddfea425 :memo: Add „Debug .gitignore files“ to git
↪docs
...
$ git diff af1a395a08221ffa83b46f562b6823cf044a108c
↪d650de52306b63b93e92bba4f15be95eddfea425
```

--staged, --cached
shows differences between the stage area and the repository.

--word-diff
shows the changed words.

\$ git restore *FILE*

changes files in the working directory to a state previously known to Git. By default, Git checks out `HEAD`, the last commit of the current branch.

Note: In Git < 2.23, `git restore` is not yet available. In this case you still need to use `git checkout`:

```
$ git checkout FILE
```

\$ git commit

makes a new commit with the added changes.

-m 'COMMIT MESSAGE'
writes a commit message directly from the command line.

--dry-run --short
shows what would be committed with the status in short format.

\$ git reset [--hard|--soft] [TARGET_REFERENCE]

resets the history to an earlier commit.

\$ git rm *PATH*

removes a file from the work and stage areas.

\$ git stash

moves the current changes from the workspace to a stash.

To be able to distinguish your hidden changes as well as possible, the following two options are recommended:

-p or --patch
allows you to partially hide changes, for example:

```
$ git stash -p
diff --git a/docs/productive/git/work.rst b/docs/productive/git/work.rst
```

(continues on next page)

(continued from previous page)

```

index cff338e..1988ab2 100644
--- a/docs/productive/git/work.rst
+++ b/docs/productive/git/work.rst
@@ -83,7 +83,16 @@
    ``list``
        lists the hidden changes.
    ``show``
-       shows the changes in the hidden files.
+       shows the changes in the hidden files, for example
...
(1/1) Stash this hunk [y,n,q,a,d,e,?]? y

```

With ? you get a complete list of options. The most common are:

Command	Description
y	Hide this change
n	Do not apply this change
q	All changes already selected will be hidden
a	Apply this and all subsequent changes
e	Edit this change manually
?	Help

branch

creates a branch from hidden files, for example:

```

$ git stash branch stash-example stash@{0}
On branch stash-example
Changes marked for commit:
  (use "git restore --staged <file>..." to remove from staging area).
   new file:   docs/productive/git/work.rst

Changes not marked for commit:
  (use "git add <file>..." to mark the changes for commit).
  (use "git restore <file>..." to discard the changes in the working directory)
   changed:   docs/productive/git/index.rst

stash@{0} (6565fdd1cc7dff9e0e6a575e3e20402e3881a82e) gelöscht

```

save MESSAGE

adds a message to the changes.

-u UNTRACKED_FILE

hides unversioned files.

list

lists the various stashes.

show

shows the changes in the stashed files.

pop

transfers the changes from the stash to the workspace and empties the stash, for example:

```
$ git stash pop stash@{2}
```

drop

empties a specific stash, for example:

```
$ git stash drop stash@{0}
stash@{0} (defcf56541b74a1ccfc59bc0a821adf0b39eaaba) deleted
```

clear

delete all your hiding places.

Review**log****See also:**

- [Git's Database Internals III: File History Queries](#)

Filter and sort

```
$ git log [-n COUNT]
```

lists the commit history of the current branch.

-n

limits the number of commits to the specified number.

```
$ git log [--after="YYYY-MM-DD"] [--before="YYYY-MM-DD"]
```

Commit history filtered by date.

Relative specifications such as `1 week ago` or `yesterday` are also permitted.

```
$ git log --author="VEIT"
```

filters the commit history by author.

It is also possible to search for several authors at the same time, for example:

```
$ git log --author="VEIT/VSC"
```

```
$ git log --grep="TERM" [-i]
```

filters the commit history for regular expressions in the commit message.

-i

ignores upper and lower case.

```
$ git log -S"FOO" [-i]
```

filters commits for specific lines in the source code.

-i

ignores upper and lower case.

```
$ git log -G"BA*"
```

filters commits for regular expressions in the source code.

```
$ git log -- PATH
```

filters the commit history for specific files.

\$ git log MAIN..FEATURE

filters for different commits in different branches, in our case between the **MAIN** and **FEATURE** branches.

However, this is not the same as `git log FEATURE..MAIN`. Let's take the following example:

```
A - B main
 \
  C - D feature
```

\$ git log MAIN..FEATURE

shows changes in **FEATURE** that are not contained in **MAIN**, that is, commits C and D.

\$ git log FEATURE..MAIN

shows changes in **MAIN** that are not contained in **FEATURE**, that is, commit B.

\$ git log MAIN...FEATURE

shows the changes on both sides, that is, commits B, C and D.

\$ git log --follow PATH/TO/FILE

This ensures that the log shows changes to a single file, even if it has been renamed or moved.

You can activate `--follow` for individual file calls by default by activating the `log.follow` option in your global configuration:

```
$ git config --global log.follow true
```

Then you no longer have to enter `--follow`, but only the file name.

\$ git log -L LINE_START_INT/LINE_START_REGEX,LINE_END_INT/LINE_END_REGEX:PATH/TO/FILE

\$ git log -L :FUNCNAME_REGEX:PATH/TO/FILE

With the `-L` option, you can perform a refined search by checking the log of only part of a file. This function allows you to thoroughly search through the history of a single function, class or other code block. It is ideal for finding out when something was created and how it was changed so that you can correct, refactor or delete it with confidence.

For more comprehensive investigations, you can also track multiple blocks. You can use multiple `-L` options at once.

\$ git log --reverse

The log usually displays the latest commit first. You can reverse this with `--reverse`. This is particularly useful if you are analysing with the `-S` and `-G` options already mentioned. By reversing the order of the commits, you can quickly find the first commit that added a specific string to the codebase.

View

\$ git log --stat --patch|-p

--stat

A summary of the number of changed lines per file is added to the usual metadata.

--patch|-p

adds the complete commit diff to the output.

\$ git log --oneline --decorate --graph --all|FEATURE

display the history graph with references, one commit per line.

--oneline

One commit per line.

--decorate

The prefixes `refs/heads/`, `refs/tags/` and `refs/remotes/` are not output.

--graph

The log usually smoothes historical branches and displays commits one after the other. This hides the parallel structure of the history when merging branches. `--graph` displays the history of the branches in ASCII format.

--all | *FEATURE*

`--all` shows the log for all branches; *FEATURE* only shows the commits of this branch.

reflog

With `git reflog`, your Git repository is not checked a second time. Instead, it displays the reference log, a record of all commits made. The reflog not only tracks changes to a branch, it also records changes to the current commit, branch changes, rebasing, etc. (et cetera) You can use it to find all unreachable commits, even those on deleted branches. This allows you to undo many otherwise destructive actions.

Let's look at the basics of using reflog and some typical use cases.

Warning: The reflog is only part of your local repository. If you delete a repository and clone it again, the new clone will have a fresh, empty reflog.

Show the reflog for HEAD**\$ git reflog**

If no options are specified, the command displays the reflog for HEAD by default. It is short for `git reflog show HEAD`. `git reflog` has other subcommands to manage the log, but `show` is the default command if no subcommand is passed.

```

1 $ git reflog
2 12bc4d4 (HEAD -> main, my-feature-branch) HEAD@{0}: merge my-feature-branch: Fast-forward
3 900844a HEAD@{1}: checkout: moving from my-feature-branch to main
4 12bc4d4 (HEAD -> main, my-feature-branch) HEAD@{2}: commit (amend): Add my feature and...
   ↪more
5 982d93a HEAD@{3}: commit: Add my feature
6 900844a HEAD@{4}: checkout: moving from main to my-feature-branch
7 900844a HEAD@{5}: commit (initial): Initial commit

```

- The output is quite dense.
- Each line is a reflog entry, the most recent first.
- The lines start with the abbreviated SHA of the corresponding commit, for example `12bc4d4`.
- The first entry is what HEAD currently refers to: `(HEAD -> main, my-feature)`.
- The names `HEAD@{N}` are alternative references for the specified commits. N is the number of returning reflog entries.
- remaining text describes the change. Above you can see several types of entries:
 - `commit: MESSAGE` for commits
 - `commit (amend): MESSAGE` for a commit change

- checkout: moving from SRC TO DST for a branch change

There are many other possible types of entries. The text should be descriptive enough that you can understand the process without looking it up in the documentation. In most cases, you will want to look through such reflog entries to find the corresponding commit SHA.

Show the reflog for a branch

You can focus on entries for a single branch by using the explicit subcommand `show` and the branch name:

```
$ git reflog show my-feature-branch
12bc4d4 (HEAD -> main, my-feature-branch) my-feature-branch@{0}: commit (amend): Add my_
↪feature and more
982d93a my-feature-branch@{1}: commit: Add my feature
900844a my-feature-branch@{2}: branch: Created from HEAD
```

Show timestamps of the entries

If you need to distinguish between similarly titled changes, the timestamps can help. For relative timestamps you can use `--date=relative`:

```
$ git reflog --date=relative
12bc4d4 (HEAD -> main, my-feature) HEAD@{vor 37 Minuten}: merge my-feature-branch: Fast-
↪forward
900844a HEAD@{vor 37 Minuten}: checkout: moving from my-feature-branch to main
12bc4d4 (HEAD -> main, my-feature-branch) HEAD@{vor 37 Minuten}: commit (amend): Add my_
↪feature and more
982d93a HEAD@{vor 38 Minuten}: commit: Add my feature
900844a HEAD@{vor 39 Minuten}: checkout: moving from main to my-feature-branch
900844a HEAD@{vor 40 Minuten}: commit (initial): Initial commit
```

And for absolute timestamps you can also use `--date=iso`:

```
$ git reflog --date=iso
12bc4d4 (HEAD -> main, my-feature) HEAD@{2024-01-11 15:26:53 +0100}: merge my-feature-
↪branch: Fast-forward
900844a HEAD@{2024-01-11 15:26:47 +0100}: checkout: moving from my-feature-branch to main
12bc4d4 (HEAD -> main, my-feature-branch) HEAD@{2024-01-11 15:26:11 +0100}: commit_
↪(amend): Add my feature and more
982d93a HEAD@{2024-01-11 15:25:38 +0100}: commit: Add my feature
900844a HEAD@{2024-01-11 15:24:37 +0100}: checkout: moving from main to my-feature-branch
900844a HEAD@{2024-01-11 15:23:56 +0100}: commit (initial): Initial commit
```


Passes all options that `git log` supports

`git reflog show` has the same options as `git log`. For example, you can use `--grep` to search for commit messages that mention *my feature* without case-sensitivity:

```
$ git reflog -i --grep 'my feature'
12bc4d4 (HEAD -> main, my-feature-branch) HEAD@{0}: merge my-feature: Fast-forward
12bc4d4 (HEAD -> main, my-feature-branch) HEAD@{2}: commit (amend): Add my feature and
↪ more
982d93a HEAD@{3}: commit: Add my feature
```

Note the expiry of entries

Reflog entries expire after a certain time when Git runs the automatic GC (garbage collection) process for your repository. This expiration time is controlled by two `gc.*` options:

`gc.reflogExpire`

The general expiration time, which is set to 90 days by default.

`gc.reflogExpireUnreachable`

The expiry time for entries relating to commits that can no longer be reached is set to 30 days by default.

You can increase these options to a longer time frame, but this is rarely useful.

Git tags

Git tags are references that point to specific commits in the Git history. This allows certain points in the history to be marked for a particular version, for example `v3.9.16`. Tags are like *Git branches* that do not change, so have no further history of commits.

`$ git tag TAGNAME`

creates a tag, where *TAGNAME* is a semantic label for the current state of the Git repository. Git distinguishes between two different types of tags: annotated and lightweight tags. They differ in the amount of associated metadata.

Annotated tags

They store not only the *TAGNAME*, but also additional metadata such as the name and email address of the person who created the tag and the date. In addition, annotated tags have messages, similar to commits. You can create such tags, for example with `git tag -a v3.9.16 -m 'Python 3.9.16'`. You can then display this additional metadata for example with `git show v3.9.16`.

Lightweight tags

Lightweight tags can be created, for example, with `git tag v3.9.16` without the `-a`, `-s` or `-m` options. They create a tag checksum that are stored in the `.git/` directory of your repo.

`$ git tag`

lists the tags of your repo, for example:

```
v0.9.9
v1.0.1
v1.0.2
v1.1
...
```

```
$ git tag -l 'REGEX'
```

lists only tags that match a regular expression.

```
$ git tag -a TAGNAME COMMIT-SHA
```

creates a tag for a previous commit.

The previous examples create tags for implicit commits that reference HEAD. Alternatively, `git tag` can be passed the reference to a specific commit that you get with [Review](#).

However, if you try to create a tag with the same identifier as an existing tag, Git will give you an error message, for example Fatal: tag 'v3.9.16' already exists. If you try to tag an older commit with an existing tag, Git will give the same error.

In case you need to update an existing tag, you can use the `-f` option, for example:

```
$ git tag -af v3.9.16 595f9ccb0c059f2fb5bf13643bfc0cdd5b55a422 -m 'Python 3.9.16'
Tag 'v3.9.16' updated (was 4f5c5473ea)
```

```
$ git push origin TAGNAME
```

Sharing tags is similar to pushing branches: by default, `git push` does not share tags, but they must be explicitly passed to `git push` for example:

```
$ git tag -af v3.9.16 -m 'Python 3.9.16'
$ git push origin v3.9.16
Counting objects: 1, done.
Writing objects: 100% (1/1), 161 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:python/cpython.git
 * [new tag]          v3.9.16 -> v3.9.16
```

To push multiple tags at once, pass the `--tags` option to the `git push` command. Others get the tags on `git clone` or `git pull` of the repo.

With `git push --follow-tags` you can also share the corresponding annotated tags with a commit.

Note: `--follow-tags` works for annotated tags, not for lightweight tags.

If you want to use `--follow-tags` for all future pushes, you can configure this with

```
$ git config --global push.followTags true
```

See also:

- `git push --follow-tags`
- `git config push.followTags`

```
$ git checkout TAGNAME
```

switches to the state of the repo with this tag and detaches HEAD. This means that any changes made now will not update the tag, but will end up in a detached commit that cannot be part of a branch and will only be directly accessible via the SHA hash of the commit. Therefore, a new branch is usually created when such changes are to be made, for example with `git checkout -b v3.9.17 v3.9.16`.

```
$ git tag -d TAGNAME
```

deletes a tag, for example:

```
$ git tag -d v3.9.16
$ git push origin --delete v3.9.16
```

Git branches

\$ git branch [-a] [-l "GLOB_PATTERN"]

shows all local branches in a repository.

-a

also shows all removed branches.

-l

restricts the branches to those that correspond to a specific pattern.

\$ git branch --sort=-committerdate

sorts the branches according to the commit date.

You can also use `git config --global branch.sort -committerdate` to make this setting your default setting.

\$ git branch BRANCH_NAME

creates a new branch based on the current HEAD.

\$ git switch [-c] BRANCH_NAME

switches between branches.

-c

creates a new branch.

Note: In Git < 2.23, `git switch` is not yet available. In this case you still need to use `git checkout`:

\$ git checkout [-b] [BRANCH_NAME]

changes the working directory to the specified branch.

-b

creates the specified branch if it does not already exist.

\$ git merge FROM_BRANCH_NAME

connects the given branch with the current branch you are currently in, for example:

```
$ git checkout main
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 setup.py | 1 -
1 files changed, 0 insertions(+), 1 deletions(-)
```

Fast forward

means that the new commit immediately followed the original commit and so the branch pointer only had to be continued.

In other cases the output can look like this:

```
$ git checkout main
$ git merge 'my-feature'
Merge made by recursive.
 setup.py | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```

recursive

is a merge strategy that is used when the merge is only to be done to HEAD.

Merge conflicts

Occasionally, however, Git runs into issues with merging, such as:

```
$ git merge 'my-feature'
Auto-merging setup.py
CONFLICT (content): Merge conflict in setup.py
Automatic merge failed; fix conflicts and then commit the result.
```

The history can then look like this, for example:

```
* 49770a2 (HEAD -> main) Fix merge conflict with my-feature
|\
| * 9412467 (my-feature) My feature
* | 46ab1a2 Hotfix directly in main
|/
* 0c65f04 Initial commit
```

See also:

- [Git Branching - Basic Branching and Merging](#)
- [Git Tools - Advanced Merging](#)

rerere to reuse recorded conflict resolutions

RERERE (reuse recorded resolutions) makes it easier for you to have to resolve the same merge conflicts again and again. This can happen, for example, if you merge a commit into several branches or if you have to rebase a branch repeatedly. Resolving merge conflicts requires concentration and energy, and it is a waste to resolve the same conflict again and again. `git rerere` is rarely called directly, however, but is usually activated globally. It is then automatically used by `git merge`, `git rebase` and `git commit`. Its most important effect is that it adds some messages to the output of these commands. You can activate it with:

```
$ git config --global rerere.enabled true
```

Let's look at an example of `git rerere` in action. Suppose you attempt a merge and run into conflicts:

```
% git merge rerere-example
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Recorded preimage for 'README.md'
Automatic merge failed; fix conflicts and then commit the result.
```

`git rerere` wrote the third line, `Preimage for 'README.md'`, meaning that the conflict was recorded before we fixed it. If we fix the conflict now, we can proceed with the merge, in our example with:

```
$ git add README.md
$ git merge --continue
Recorded resolution for 'README.md'.
[main 5935d00] Merge branch 'rerere-example'
```

`git rerere` now reports `conflict resolution recorded for 'README.md'`, meaning that it has saved how we resolved the conflicts in this file. Suppose you undo this merge because you realise that it was not finished:

```
$ git reset --keep @~
```

Later you repeat the merging process:

```
$ git merge rerere-example
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Resolved 'README.md' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
When finished, apply stashed changes with `git stash pop`
```

`git rerere` solved the conflict using the previous solution, which means it reused your previous merge. Now check that the file is correct and then continue:

```
$ git add README.md
$ git merge --continue
[main c922b21] Merge branch 'rerere-example'
```

`git rerere` saves its data within the `.git` directory of your Git repository in an `rr-cache` directory. You should note two things here:

1. The `rerere` cache is local. It is not shared when you perform a `git push`, so your team colleagues cannot reuse the merges you have performed.
2. Git's automatic garbage collection deletes entries from the `rr-cache`. It is controlled by two configuration options:

`gc.rerereResolved`

determines how long entries for resolved conflicts are kept. The default value is 60 days. And with `git config gc.rerereResolved` you can change the default values for your project.

`gc.rerereUnresolved`

determines how long entries for unresolved conflicts are kept. The default value is 15 days.

Delete branches

```
$ git branch -d [BRANCH_NAME]
```

deletes the selected branch if it has already been transferred to another.

`-D` instead of `-d` forcing the deletion.

See also:

- [Git Branching - Branches in a Nutshell](#)

Remote branches

So far, these examples have all shown local branches. However, the `git branch` command also works with remote branches. To work with remote branches, a remote repository must first be configured and added to the local repository configuration:

```
$ git remote add origin https://ce.cusy.io/veit/NEWREPO.git
```

Add remote branches

Now the branch can also be added to the remote repository:

```
$ git push origin [BRANCH_NAME]
```

With `git branch -d` you delete the branches locally only. To delete them on the remote server as well, you can type the following:

```
$ git push --set-upstream origin [BRANCH_NAME]
```

If you want to add all branches of a local repository to the remote repo, you can do this with:

```
$ git push --set-upstream origin --all
```

You can configure the following so that this happens automatically for branches without a tracking upstream:

```
$ git config --global push.autoSetupRemote true
```

Delete remote branches

To remove remote branches locally, you can run `git fetch` with the `--prune` or `-p` option. You can also make this the default behaviour by enabling `fetch.prune`:

```
$ git config --global fetch.prune true
```

See also:

[PRUNING](#)

Rename branches

You can rename branches, for example with

```
$ git branch --move master main
```

This changes your local `master` branch to `main`. In order for others to see the new branch, you must push it to the remote server. This will make the `main` branch available on the remote server:

```
$ git push origin main
```

The current state of your repository may now look like this:

```
$ git branch -a
* main
remotes/origin/HEAD -> origin/master
remotes/origin/main
remotes/origin/master
```

- Your local `master` branch has disappeared because it has been replaced by the `main` branch.
- The `main` branch is also present on the remote computer.
- However, the `master` branch is also still present on the remote server. So presumably others will continue to use the `master` branch for their work until you make the following changes:
 - For all projects that depend on this project, the code and/or configuration must be updated.

- The test-runner configuration files may need to be updated.
- Build and release scripts need to be adjusted.
- The settings on your repository server, such as the default branch of the repository, merge rules and others, need to be adjusted.
- References to the old branch in the documentation need to be updated.
- Any pull or merge requests that target the `master` branch should be closed.

After you have done all these tasks and are sure that the `main` branch works the same as the `master` branch, you can delete the `master` branch:

```
$ git push origin --delete master
```

Team members can delete their locally still existing references to the `master` branch with

```
$ git fetch origin --prune
```

Git rebase

The commands `git rebase` and `git merge` allow you to merge *Git branches*. While `git merge` is always a moving forward change approach, `git rebase` has powerful history rewrite functions. Here we take a look at its configuration, use cases and pitfalls.

In doing so, `git rebase` moves a sequence of commits to a new base commit and can be useful for *Feature branch workflows*. Internally, Git achieves this by creating new commits and applying them to the specified base; so the same-looking commits from branches are entirely new commits.

The main reason for `git rebase` is to maintain a linear project progression. If the main branch has evolved since you started working on a feature branch, you might want to keep the latest updates to the main branch in your feature branch, but keep the history of your branch clean. This would have the advantage that you could later do a clean `git merge` of your functional branch into the main branch. This *clean history* also makes it easier for you to find a regression with *Find regressions with git bisect*. A more realistic scenario would be the following:

1. An error is found in the main branch in a function that once worked without errors.
2. With the *clean history* of the main branch, *Review* should allow for quick conclusions.
3. If *Review* does not lead to the desired result, *git bisect* will probably help. In this case, the clean Git history helps `git bisect` in the search for the regression.

Warning: The published history should only be changed in very rare exceptional cases, as the old commits would be replaced by new ones and it would look as if this part of the project history had suddenly disappeared.

See also:

[git rebase: what can go wrong?](#)

Note: `git rebase` is also covered briefly in *Jupyter Notebooks with Git* and *Feature branch workflows*.

Rebasing dependent branches with `-update-refs`

When you are working on a large feature, it is often helpful to spread the work over several branches that build on each other.

However, these branches can be cumbersome to manage if you need to overwrite the history in an earlier branch. Since each branch depends on the previous branches, rewriting commits in one branch will result in subsequent branches no longer being connected to the history.

Git 2.38 ships with a new `--update-refs` option for `git rebase` that will perform such updates for you without you having to manually update each branch and without subsequent branches losing their history.

If you want to use this option on every rebase, you can run `git config --global rebase.updateRefs true` to make Git behave as if the `--update-refs` option is always specified.

See also:

rebase: add `-update-refs` option

Delete commits with `git rebase`

This can also be easily realised with `git rebase`, whereby you do not have to delete the line in your editor but replace the line `pick` with `r` (*reword*).

```
$ git rebase -i SHA origin/main
```

`-i`

Interactive mode, in which your standard editor is opened and a list of all commits after the commit with the hash value *SHA* to be removed is displayed, for example

```
pick d82199e Update readme
pick 410266e Change import for the interface
...
```

If you now remove a line, this commit will be deleted after saving and closing the editor. Then the remote repository can be updated with:

```
$ git push origin HEAD:main -f
```

Modify a commit message with rebase

This can also be easily with `rebase` by not deleting the line in your editor but replace `pick` with `r` (*reword*).

rebase as standard `git pull` strategy

Normally, `git pull` fetches and merges new remote commits without any problems. Usually only new commits from the remote branch are added, a so-called fast-forward merge. However, if both the local and remote branches have new commits, the branches will diverge. You must then somehow harmonise the different histories. By default, as of Git 2.33.1, any discrepancy will cause `git pull` to stop and display the following message:

```
$ git pull
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
```

(continues on next page)

(continued from previous page)

```

hint: your next pull:
hint:
hint:  git config pull.rebase false  # merge
hint:  git config pull.rebase true   # rebase
hint:  git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.

```

The notes allow three options:

git config pull.rebase false

merges the local and remote commits. Before Git 2.33.1, Git always used this merge.

git config pull.rebase true

The local commits are transferred to the remote commits.

git config pull.ff only

always leads to an error with divergent branches. You can then decide on a case-by-case basis with `--no-rebase` (which means merge) or `--rebase` whether you want to merge or rebase.

Tip: I recommend `git config pull.rebase true`, as merging can be confusing. Rebasing the local commits to the remote ones makes the story linear, which is more understandable.

Make rebase part of your standard strategy:

```
$ git config --global pull.rebase interactive
```

If `git pull` then encounters divergent local and remote branches, it will perform a rebase:

```

$ git pull
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
error: could not apply e50dfe5...
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --
,→abort".
Could not apply e50dfe5...

```

Undo changes

With Git 2.23, `git restore` was added for undoing file changes. Previously, this task was performed by `git reset`, which also has other tasks:

\$ `git restore`

changes files in the working directory to a state that was previously known to Git. By default, Git HEAD checks out the last commit of the current branch.

Note:

In Git < 2.23, `git restore` is not yet available. In this case, you still need to use `git checkout`:

```
$ git checkout FILE
```

\$ `git restore [-S|--staged] PATH/TO/FILE`

undoes the addition of files. The changes are retained in your workspace so that you can change and add them again if necessary.

The command is equivalent to `$ git reset PATH/TO/FILE`.

\$ `git restore [-SW] FILE`

undoes the addition and changes in the workspace.

\$ `git restore [-s|--source] BRANCH FILE`

restores a change to the version in the *BRANCH*.

\$ `git restore [-s|--source] @~ FILE`

restores a change to the previous commit.

\$ `git restore [-p|--patch]`

lets you select the changes to be undone individually.

\$ `git reset [--hard | --mixed | --soft | --keep] TARGET_REFERENCE`

resets the history to an earlier commit.

Warning: The risk with `reset` is that work can be lost. Although commits are not deleted immediately, they can become orphaned so that there is no longer a direct path to them. They must then be found and restored promptly with *reflog* as Git usually deletes all orphaned commits after 30 days.

```
$ git reset @~
```

@~

cancels the last commit, whereby its changes are now transferred back to the stage area.

If there are changes in the stage area, these are moved to the work area, for example:

```
$ echo 'My first repo' > README.rst
$ git add README.rst
$ git status
On branch main
Changes marked for commit:
  (use "git rm --cached <Datei>..." to remove from staging area)
    New file:   README.rst
$ git reset
$ git status
```

(continues on next page)

(continued from previous page)

```
On branch main
Unversioned files:
  (use "git add <file>...", to mark the changes for commit)
  README.rst
```

@~3

takes back the last three commits.

'@{u}'takes the remote version (*upstream*) of the current branch.**--hard**

discards the changes in the staging and working area as well.

```
$ git status
On branch main
Changes marked for commit:
  (use "git rm --cached <Datei>..." to remove from staging area)
  New file:      README.rst
$ git reset --hard
$ git status
On branch main
nothing to commit (create/copy files and use "git add" to version)
```

--mixed

resets the stage area, but not the work area, so that the changed files are retained but not marked for commit.

Tip: I usually prefer `--soft` over `--mixed`: it keeps the undone changes separate so that any additional changes are explicit. This is especially useful if you have changes to the same file in the stage and workspace.

--soft

takes back the commits, but leaves the stage and workspace unchanged.

--keepresets the stage area and updates the files in the work area that differ between COMMIT and HEAD, but retains those that differ between stage and work area, these are files with changes that have not yet been added. If a file that differs between COMMIT and stage area has unadded changes, `reset` will be cancelled.You can then deal with your uncommitted changes, perhaps undoing them with `git restore` or hiding them with `git stash`, before trying again.

Tip: Many other guides recommend `--hard` for this task, probably because this mode has been around for a while. However, this mode is riskier because it irrevocably discards the changes not included in the commit without asking questions. However, I use `--keep` and if I want to discard all uncommitted changes before the `reset`, I use `git restore -SW`.

\$ git revert COMMIT_SHA

creates a new commit and reverts the changes of the specified commit so that the changes are inverted.

\$ git fetch --prune REMOTE

Remote refs are removed when they are removed from the remote repository.

\$ git commit --amend

updates and replaces the last commit with a new commit that combines all deployed changes with the contents

of the previous commit. If nothing is provided, only the previous commit message is rewritten.

Reference for common reset commands

Undo all local changes to a branch

```
$ git reset --keep '@{u}'
```

Undo all commits in the current branch

`git merge-base` selects the commit where two branches have split. Pass `@` and `main` to select the commit where the current branch is forked from `main`. Reset it to undo all commits on the local branch with:

```
$ git reset --soft $(git merge-base @ main)
```

Undo all changes in the current branch

```
$ git reset --keep main
```

Undo commit in the wrong branch

If you have accidentally committed to an existing branch instead of creating a new branch first, you can change this in the following three steps:

1. create a new branch with `$ git branch NEW_BRANCH`
2. Resets the last commit in your active branch with `$ git reset --keep @~`
3. Apply the changes to the new branch with `$ git switch NEW_BRANCH`

Restoring a deleted branch

Assuming you have accidentally deleted an unmerged branch, you can recreate the branch with the corresponding SHA:

```
$ git branch -D new-feature  
Branch new-feature entfernt (war d53e431).
```

The output contains the SHA commit to which the branch pointed. You can recreate the branch with this SHA:

```
$ git branch new-feature d53e431
```

But what if you have deleted the branch and the corresponding terminal history has been lost? To find the SHA again, you can pass the `reflog` output to `grep`:

```
$ git reflog | grep -A 1 new-feature  
12bc4d4 HEAD@{0}: checkout: moving from new-feature to main  
d53e431 HEAD@{1}: commit: Add new feature  
12bc4d4 HEAD@{2}: checkout: moving from main to new-feature  
12bc4d4 HEAD@{3}: merge my-feature: Fast-forward
```

-A 1 displays an additional line after each hit. The output shows several *reflog* entries that refer to the branch. The first entry shows a change from `new-feature` to `main`, with the commit SHA on `main`. The entry before it is the last change to `new-feature` with the SHA to restore:

```
$ git branch triceratops-enclosure 43f66f9
```

By default, you can save such a branch within 30 days after deleting the branch, as `gc.reflogExpireUnreachable` is usually set to do so.

Undoing a commit change

Let's return to the introductory example. Imagine you have made a commit and changed it later. Then you realise that the change should be undone. How can you proceed? If you can still see the original Git commit output in your terminal history, you can retrieve the SHA from there and undo the change. But if this is no longer possible, it's time for the *reflog*. Check the *reflog* for the branch:

```
$ git reflog my-feature-branch
12bc4d4 (HEAD -> main, my-feature-branch) my-feature-branch@{0}: commit (amend): Add my_
↪feature and more
982d93a my-feature-branch@{1}: commit: Add my feature
900844a my-feature-branch@{2}: branch: Created from HEAD
```

The first entry, `commit (amend)`, shows the creation of the amended commit. The second entry shows the original commit, which we now want to return to with a hard reset:

```
$ git reset --hard 982d93a
```

You may then want to restore the content of the changed commit in order to correct it and change it again. Do this with `git restore` from the changed commit SHA, which is at the top of the previous *reflog* output:

```
$ git restore -s 12bc4d4
```

Undoing a faulty rebase

Imagine you are working on a `new-feature` branch with three commits, of which you want to undo the middle one:

```
$ git rebase -i main
```

```
pick d53e431 Add new feature
-pick 329271a More performant implementation for the new feature
-pick 1d6c477 Add API docs
```

However, you have now inadvertently deleted the last commit. If you can no longer see the SHA value in the terminal history, you can pass the *reflog* output to `grep` again:

```
$ git reflog| grep 'API docs'
1d6c477 HEAD@{2}: commit: Add API docs
```

With this SHA, the commit can now be restored with *Git cherry-pick*:

```
$ git cherry-pick 1d6c477
```

Remove a file from the history

A file can be completely removed from the current branches Git history. This could be necessary if you accidentally committed passwords or huge files:

```
$ git filter-repo --invert-paths --path path/somefile
$ git push --no-verify --mirror
```

Note: Inform the team members that they should create a clone of the repository again.

Remove a string from the history

```
$ git filter-repo --message-callback 'return re.sub(b"^git-svn-id:.*\n", b"", message,
↩flags=re.MULTILINE)'
```

See also:

- [git-filter-repo — Man Page](#)
- [git-reflog](#)
- [git-gc](#)

Git best practices

Commit early

Make your first commit after you've finished the initial installation and before you make your first changes. For a cookie cutter template, for example, proceed as follows:

```
$ pipenv run cookiecutter https://github.com/veit/cookiecutter-namespace-template.git
full_name [Veit Schiele]:
email [veit@cusy.io]:
github_username [veit]:
project_name [cusy.example]:
...
```

These initial changes can then be checked in with:

```
$ cd cusy.example
$ git init
$ git add *
$ git add .gitignore
$ git commit -m 'Initial commit'
$ git remote add origin ssh://git@github.com:veit/cusy.example.git
$ git push -u origin main
```

Exclude undesired files

Temporary files, jupyter checkpoint folders and builds have no business in a git repository. Credentials do not either. The `.gitignore` file contains a list of paths that git will not add unless you ask for it explicitly.

You can find a template `.gitignore` file for Python projects in the [dotfiles](#) repository. The [gitignore.io](#) website contains `.gitignore` files for other programming languages. The `.gitignore` file itself should be checked in, too:

```
$ git add .gitignore
$ git commit -m 'add .gitignore file'
```

If you have accidentally checked undesired files into your Git repository, you can remove them again with:

```
$ git rm -r .ipynb_checkpoints/
```

Write a README

Each repository should also have a `README.rst` file that describes the deployment and the basic structure of the code.

Commit often

Each completed task and subtask should be immediately followed by a commit. Incomplete work also may be stored on git. As a rule of thumb you should commit at least daily before leaving work. In busy times it is common to commit every 10 minutes.

Frequent commits make it easier for you to:

- isolate errors
- understand the code
- maintain the code in the future

If you have made several changes to a file, you can split them up into several commits later with:

```
$ git add -p my-changed-file.py
```

Don't change the published history

Even if you later find out that a commit that has already been published with `git push` contains one or more errors, you should never try to undo this commit. Rather, you should fix the error that have occurred through further commits.

Warning: Workflows with `git rebase` are a reasonable exception to this rule.

Choose a Git workflow

Choose a workflow that fits best to your project. Projects are by no means identical and a workflow that fits one project does not necessarily have to fit in another project. A different workflow can be recommended initially than in the further progress of the project.

Write meaningful commit messages

By creating insightful and descriptive commit messages, you make working in a team a lot easier. They allow others to understand your changes. They are also helpful at a later point in time to understand which goal should be achieved with the code.

Usually short messages, 50–72 characters long, should be specified and displayed on one line, eg with `git log --oneline`.

With `git blame` you can later specify for each line in which revision and by which author the change was made. You can find more information on this in the Git documentation: [git-blame](#).

If you use gitmojis in your commit messages, you can easily see the intent of the commit later.

Note:

- [gitmoji.dev](#)
 - [github.com/carloscuesta/gitmoji](#)
 - [github.com/carloscuesta/gitmoji-cli](#)
 - Visual Studio Code Extension
-

GitLab also interprets certain commit messages as links, for example:

```
$ git commit -m "Awesome commit message (Fix #21 and close group/otherproject#22)"
```

- links to issues: `#NUMBER`
- links to issues in other projects: `GROUP/PROJECT#NUMBER`
- links to merge requests: `!NUMBER`
- links to snippets: `$NUMBER`

There should be at least one ticket for each commit that should provide more detailed information about the changes.

There should be at least one ticket for each commit, which should contain more detailed information about the changes. Alternatively, you can also write multi-line commit messages containing this information, for example with:

```
$ git commit -m 'Expand section on meaningful commit messages' -m 'Fix the serious_↵
↵problem'
```

Or, if you just enter `git commit`, your editor will open, for example with the following text:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
```


Git expects you to insert your commit message at the beginning of the file. After you have finished editing the file, Git reads its contents and continues. It cleans up the file by removing lines commented with `#` and subsequent empty lines. If the message is empty after cleaning up, Git cancels the commit – this is useful if you realise that you have forgotten something. Otherwise, the commit is created with the remaining content. However, GitLab uses `#` as a prefix for the number of an item. This double meaning of `#` can lead to confusion if you write a commit message that refers to an item:

```
Expand section on meaningful commit messages
#21: Add multi-line commit messages
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Changes to be committed:
#   modified:   productive/git/best-practices.rst
#
```

Git usually removes the line starting with `#21` so that the message looks like this:

```
Expand section on meaningful commit messages
```

Avoid this mishap by using an alternative clean-up mode called *Scissors*. You can activate it globally with:

```
$ git config --global commit.cleanup scissors
```

Git then starts each new commit message with the *Scissors* line:

```
# ----- >8 -----
# Do not modify or remove the line above.
# Everything below it will be ignored.
#
# On branch main
# ...
#
```

Specify co-authors

If you are working on a commit with a team member, it's good to acknowledge their contribution with the co-authored-by trailer. Trailers are additional metadata at the end of the commit message that use a *KEY: VALUE* syntax and can be repeated to list multiple values:

```
Expand section on meaningful commit messages
#21: Add multi-line commit messages
co-authored-by: Kristian Rother <kristian.rother@cusy.io>
co-authored-by: Frank Hofmann <frank.hofmann@cusy.io>
```

GitLab analyses the co-authored-by lines to display all avatars of the commit and also to update the profile statistics of the co-authors, etc..

Maintain your repository regularly

You should perform the following maintenance work regularly:

Validate the repo

The command `git fsck` checks whether all objects in the internal datastructure of git are consistently connected with each other.

Compresses the repo

Save storage space with the command `git gc` or `git gc --aggressive`.

See also:

- `git gc`
- [Git Internals - Maintenance and Data Recovery](#)

Clean up remote tracking branches

Unused branches on a server can be removed with `git remote update --prune`. It is even better if you change the default setting so that remotely deleted branches are also deleted locally with `git fetch` and `git pull`. You can achieve this with:

```
$ git config --global fetch.prune true
```

Check forgotten work

Display a list of saved stashes with `git stash list`. They can be removed with `git stash drop`.

Check your repositories for unwanted files

With [Gitleaks](#) you can regularly check your repositories for unintentionally saved access data.

You can also run Gitleaks automatically as a GitLab action. To do this, you need to include the [Secret-Detection.gitlab-ci.yml](#) template, for example, in a stage called `secrets-detection` in your `.gitlab-ci.yml` file:

```
include:  
  - template: Security/Secret-Detection.gitlab-ci.yml
```

The template creates secret detection jobs in your CI/CD pipeline and searches the source code of your project for secrets. The results are saved as a [Secret Detection Report Artefakt](#) that you can download and analyse later.

See also:

- [GitLab Secret Detection](#)

With [git-filter-repo](#) you can remove unwanted files from your Git history.

Git workflows

Here, Git workflow is understood as a recommendation for using Git to enable a consistent and efficient way of working. Since Git makes branching and merging much easier compared to older versioning systems like SVN, this allows for a variety of different workflows and there is no one ideal process for best interacting with Git.

However, all of the workflows presented expect everyone on the team to use the same workflow for changes. Therefore, at the outset, a team should collectively agree on a particular Git workflow that they feel is most appropriate for that project. Size and team culture play a role in keeping the complexity of the workflow and the number of errors as low as possible.

In the following, we discuss some of these Git workflows.

Git Flow

Git Flow was one of the first proposals for the use of Git branches. It recommended a `main` branch and a separate `develop` branch as well as various other branches for features, releases and hotfixes. The various developments should be brought together in the `develop` branch, then transferred to the `release` branch and finally end up in the `main` branch.

Drawbacks of Git Flow

While Git Flow is a well-defined but complex standard, it creates two practical problems:

- Most developers and tools assume that the `main` branch is the branch from which branches and merges are executed. With Git Flow, there is additional work involved because you always have to switch to the `develop` branch first.
- The `hotfixes` and `release` branches also bring additional complexity, which should only bring advantages in the rarest of cases.

In response to the problems of Git Flow, [GitHub](#) and [Atlassian](#) developed simpler alternatives that are mostly limited to so-called *Feature branch workflows*.

See also:

[Vincent Driessen: A successful Git branching model](#)

First steps

Git-flow is just an abstract idea of a git workflow, where the branches and the merges are given. There is also software, `git-flow`, to assist with this workflow.

Installation

```
$ wget -q -O - --no-check-certificate https://github.com/nvie/gitflow/raw/develop/  
↪contrib/gitflow-installer.sh | bash
```

```
$ sudo apt install git-flow
```

```
$ brew install git-flow
```

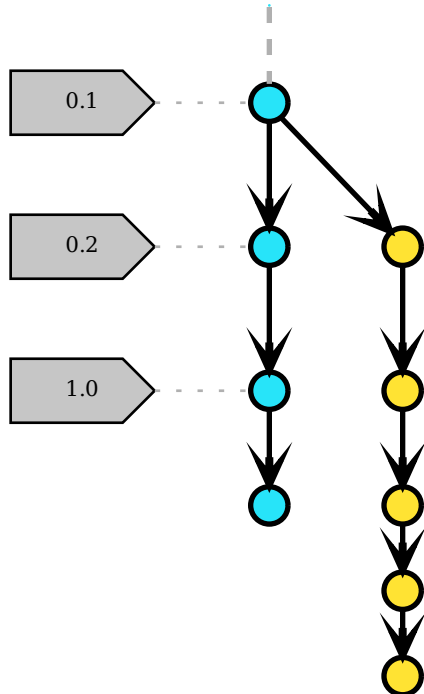
Initialise

`git-flow` is a wrapper for Git. The `git flow init` command not only initiates a directory, but also creates branches for you:

```
$ git flow init
Initialized empty Git repository in /home/veit/my_repo/.git/
No branches exist yet. Base branches must be created now.
Branch name for production releases: [master] main
Branch name for "next release" development: [develop]
How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [.git/hooks]
```

Alternatively, you could have entered the following:

```
$ git branch develop
$ git push -u origin develop
```



This workflow provides two branches to record the history of the project:

main

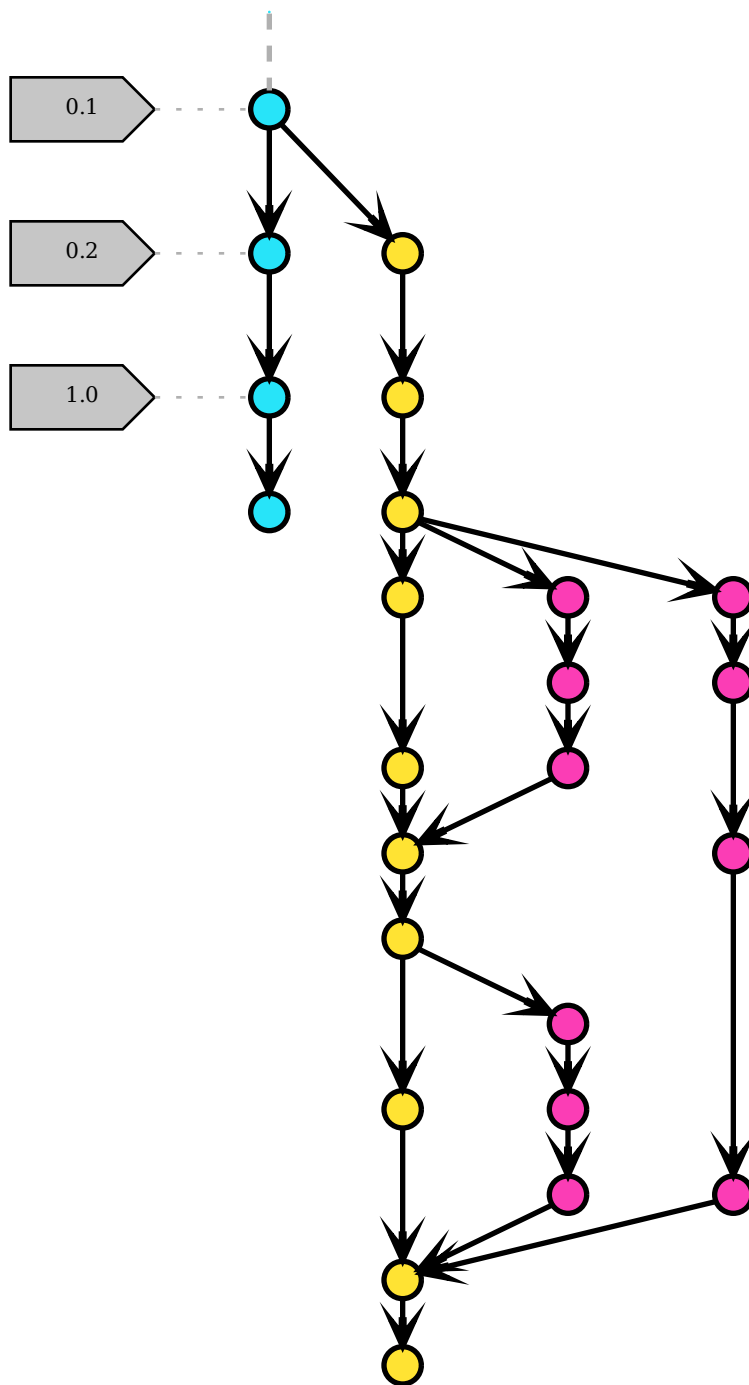
contains the official release history, and all commits in this branch should be tagged with a version number.

develop

integrates the features.

Feature branches

Each new feature should be created in its own branch, which can be pushed to the remote repository at any time. However, a feature branch is not created from the `main` branch but from the `develop` branch; and when a feature is finished, it is also merged back into the `develop` branch.



You can create such feature branches with `git flow`:

```
$ git flow feature start 17-some-feature
```

(continues on next page)

(continued from previous page)

```
Switched to a new branch 'feature/17-some-feature'
```

```
Summary of actions:
```

- A new branch 'feature/17-some-feature' was created, based on 'develop'
- You are now on branch 'feature/17-some-feature'

```
...
```

... or with

```
$ git switch -c feature/17-some-feature
Switched to a new branch 'feature/17-some-feature'
```

Conversely, you can complete your feature branch with

```
$ git flow feature finish 17-some-feature
Switched to branch 'develop'
Already up to date.
Deleted branch feature/17-some-feature (was a2d223f).
...
```

... or with

```
$ git switch develop
$ git merge feature/17-some-feature
$ git branch -d feature/17-some-feature
Deleted branch feature/17-some-feature (was a2d223f).
```

Release branches

If the `develop` branch contains enough features for a release or a fixed release date is approaching, a `release` branch is created from the `develop` branch, to which no new features should be added from this point on, but only bug fixes and changes related to this release. If the release can be delivered, the `release` branch is on the one hand merged into the `main` branch and tagged with a version number, and on the other hand merged back into the `develop` branch, which may have developed further since the creation of the `release` branch.

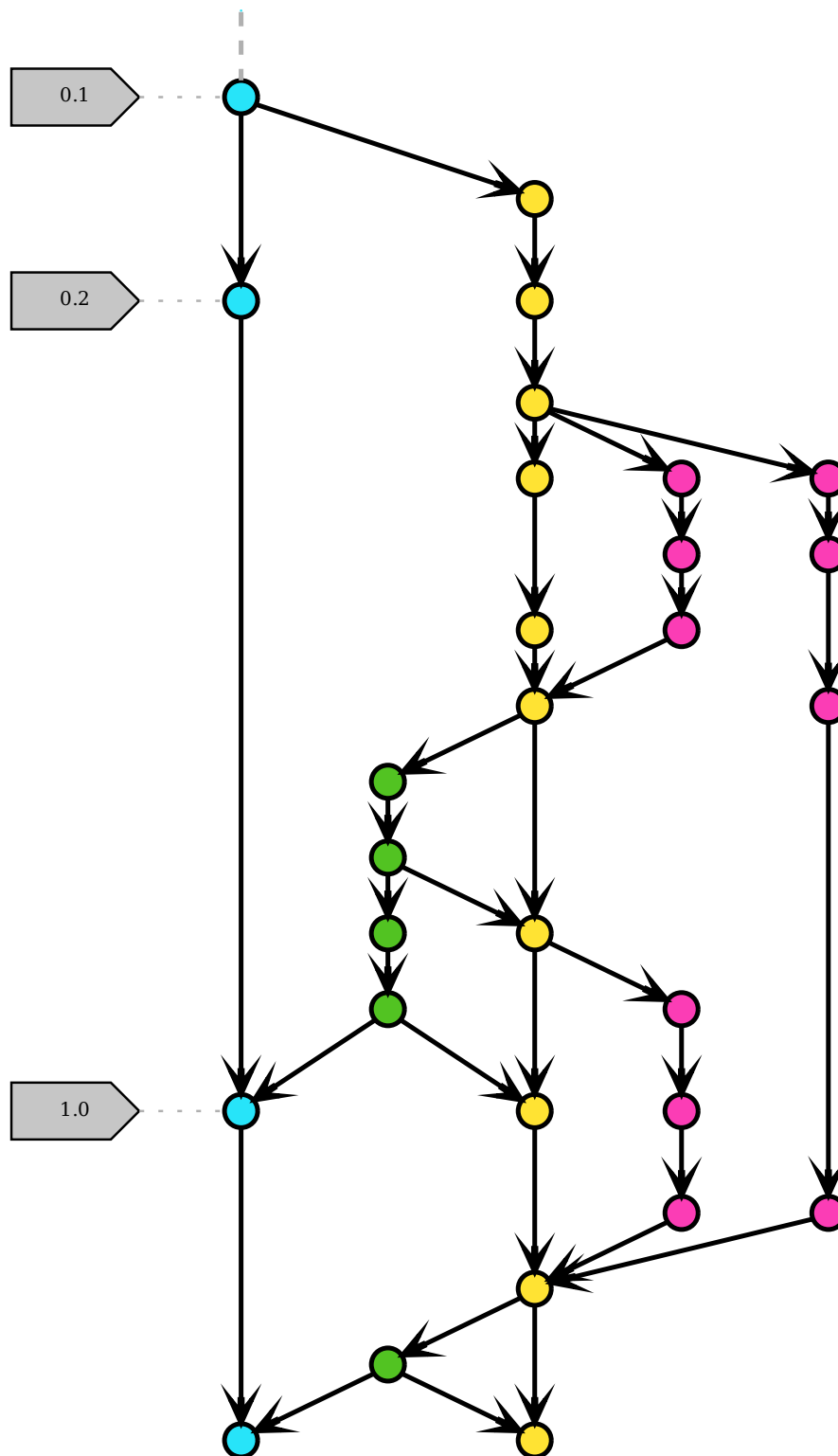
```
$ git flow release start 0.1.0
Switched to a new branch 'release/0.1.0'
...
$ git flow release finish '0.1.0'
Switched to branch 'main'
Deleted branch release/0.1.0 (was a2d223f).
```

```
Summary of actions:
```

- Release branch 'release/0.1.0' has been merged into 'main'
- The release was tagged '0.1.0'
- Release tag '0.1.0' has been back-merged into 'develop'
- Release branch 'release/0.1.0' has been locally deleted
- You are now on branch 'develop'

... or

```
$ git switch develop
$ git branch develop/0.1.0
...
$ git switch main
$ git merge release/0.1.0
$ git tag -a 0.1.0
$ git switch develop
$ git merge release/0.1.0
$ git branch -d release/0.1.0
```

Hotfix branches

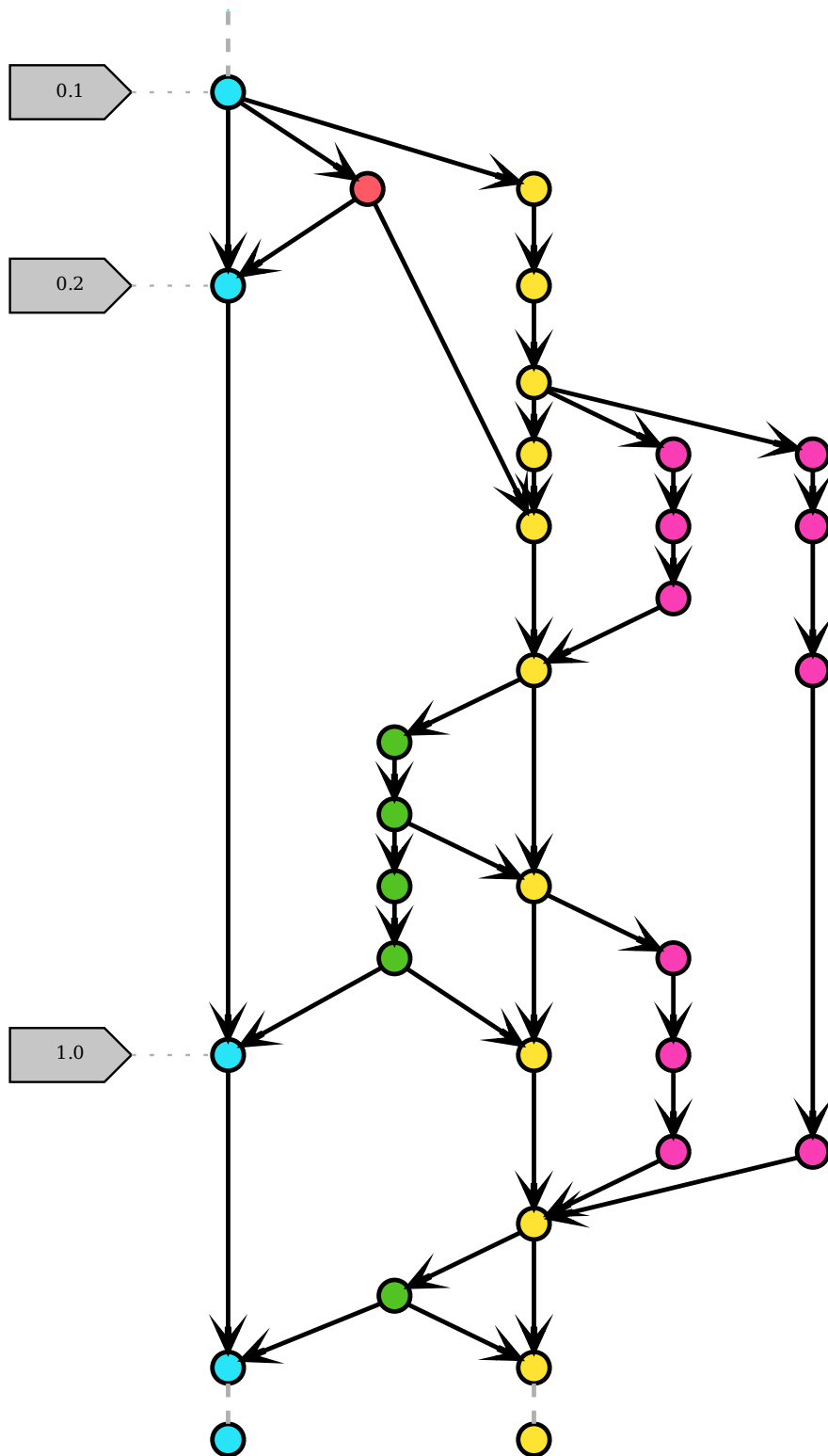
Hotfix branches are suitable for quick patches of production versions. They are similar to release branches and feature branches, but are based on the main branch instead of the develop branch. This makes it the only branch that should be forced directly from the main branch. Once the hotfix has been completed, it should be merged into both the main and develop branches and, if necessary, into the current release branch. The main branch should also be tagged with a new version number.

```
$ git flow hotfix finish 37-some-bug
Switched to branch 'develop'
Merge made by the 'recursive' strategy.
...
Deleted branch hotfix/37-some-bug (was a2d223f).

Summary of actions:
- Hotfix branch 'hotfix/37-some-bug' has been merged into 'main'
- The hotfix was tagged '0.2.0'
- Hotfix tag '0.2.0' has been back-merged into 'develop'
- Hotfix branch 'hotfix/37-some-bug' has been locally deleted
- You are now on branch 'develop'
```

... or

```
$ git switch main
Switched to branch 'main'
...
$ git merge hotfix/37-some-bug
$ git tag -a 0.2.0
$ git switch develop
$ git merge hotfix/37-some-bug
$ git branch -d hotfix/37-some-bug
```



Feature branch workflows

The basic idea behind feature branch workflows is that the development of individual features should take place in a dedicated branch and not in the `main` branch. This encapsulation facilitates the work in a development team, as changes in the `main` branch do not disturb and can initially be neglected. Conversely, this should prevent the `main` branch from being contaminated by unfinished code. This second argument then also facilitates [continuous integration](#) with other components.

See also:

- [Feature Driven Development](#)
- Martin Fowler: [Feature Branch](#)

Merge or pull requests

Encapsulating the development of individual features in a branch also allows you to use merge or pull requests to discuss changes with others in the team and give them the opportunity to approve a feature before it is integrated into the official project. However, if you encounter problems in your feature development, you can also use merge or pull requests to discuss possible solutions with others in the team.

Merge or pull requests are provided by web-based services such as [GitHub](#), [GitLab](#) and [Atlassian](#) for reviewing and commenting on changes. You can also use `@ID` in your comments to ask specific people on the project team directly for feedback. If you use automated testing, you can also see the test results here; perhaps the coding style does not correspond to your project guidelines, or the test coverage is insufficient. In the merge or pull requests, such discussions are encouraged and documented without appearing directly as commits in the repository.

Warning: Merge or pull requests are not part of Git itself, but of the respective web-based service. They are also not standardised, so that they can only be transferred with difficulty when switching to another service.

See also:

- [About pull requests](#)
- [Making a Pull Request](#)
- [Merge requests](#)

GitHub Flow

[GitHub Flow](#) was intended to be a greatly simplified alternative to [Git Flow](#), with only different feature branches in addition to the `main` branch. The lifecycle of a Git feature branch could then look like this:

1. All feature branches start on the basis of the current `main` branch.

To do this, we first switch to the `main` branch, get the latest changes from the server and update our local copy of the repository:

```
$ git switch main
$ git fetch origin
$ git reset --hard origin/main
```

2. Creating the feature branch.

We create a feature branch with `git switch -c` and the number of the ticket in the task management that describes this feature.

```
$ git switch -c 17-some-feature
```

3. Add and commit changes.

```
$ git add SOMEFILE
$ git commit
```

4. Push the feature branch with the changes.

By pushing the feature branch with your changes, you not only create a backup copy of your changes, but you also allow others in the team to view the changes.

```
$ git push -u origin 17-some-feature
```

The `-u` parameter adds the `17-some-feature` branch to the upstream Git server (`origin`) as a remote branch. In the future, you can push into this branch without having to specify any further parameters.

5. Make a merge or pull request

Once you have completed a feature, it is not immediately merged into the `main` branch, but a merge or pull request is created, giving others in the development team the opportunity to review your changes. Any changes to this branch will now also be reflected in this merge or pull request.

6. Merge

Once your merge or pull request is accepted, you must first ensure that your local `main` branch is synchronised with the upstream `main` branch; only then can you merge the feature branch into the `main` branch and finally push the updated `main` branch back into the upstream `main` branch. However, this will not infrequently lead to a merge commit. Nevertheless, this workflow has the advantage that a clear distinction can be made between feature development and merging.

Simple Git workflow

Atlassian also recommends a [similar strategy](#), but they recommend *rebasing* the feature branches. This gives you a linear progression by moving the changes in the feature branch to the top of the `main` branch before merging with a fast-forward merge.

1. Use `rebase` to keep your feature branch up to date with `main`:

```
$ git fetch origin
$ git rebase -i origin/main
```

In the rare case that others from the team are also working in the same feature branch, you should also adopt their changes:

```
$ git rebase -i origin/17-some-feature
```

Resolves any conflicts arising from `rebase` at this stage. This should have resulted in a number of clean merges by the end of feature development. It also keeps the history of your feature branches clean and focused, without distracting noise.

2. When you are ready for feedback, push your branch:

```
$ git push -u origin 17-some-feature
```

You can then make a merge or pull request.

After this push, you can always update the remote branch in response to feedback.

3. After the review is complete, you should do a final clean-up of the feature branch's commit history to remove unnecessary commits that do not provide relevant information.
4. When development is complete, merge the two branches with `-no-ff`. This will preserve the context of the work and make it easy to revert the entire feature if needed:

```
$ git switch main
$ git pull origin main
$ git merge --no-ff 17-some-feature
```

The simple-git-workflow using `rebase` creates a strictly linear version history. In this linear history it is easier to understand changes over time and to find bugs with *bisect*.

Summary

The main advantages of feature branches workflows are as follows

- Features are isolated in individual branches so that each team member can work independently.
- At the same time, team collaboration is enabled via merge or pull requests.
- The code inventory to be managed remains relatively small because the feature branches can usually be quickly transferred to the `main`.
- The workflows correspond to the usual methods of continuous integration.

However, they cannot answer how deployments to different environments or splitting into different releases should be done. Possible answers to this are described in *Deployment and release branches*.

See also:

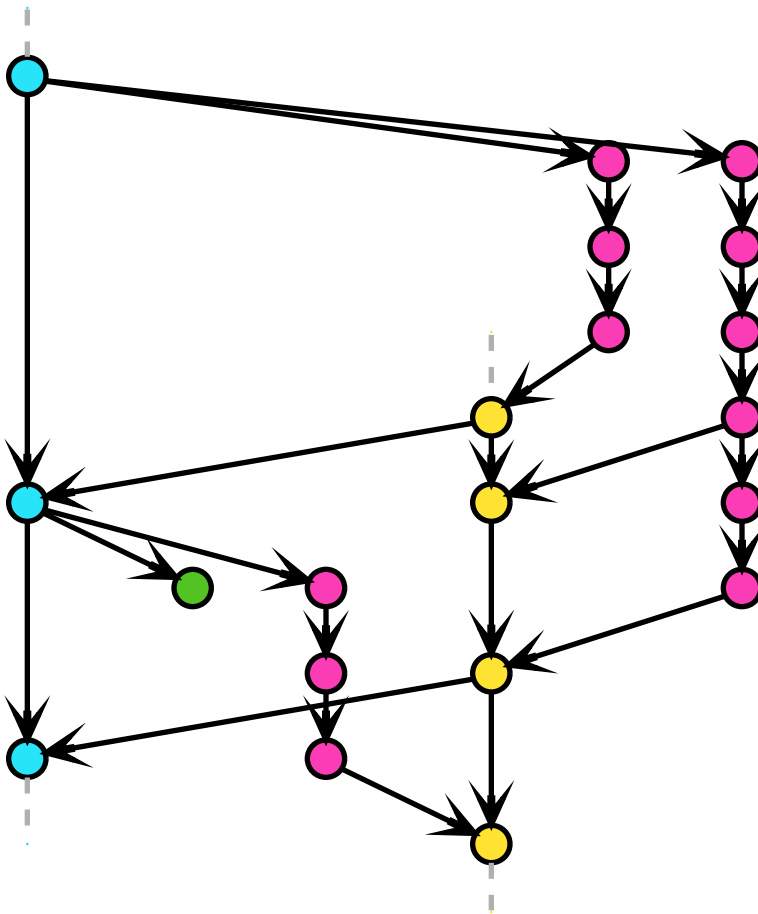
Both variations of feature branches are simpler alternatives of the considerably more complex *Git Flow*.

Deployment and release branches

Deployment branches

We recommend one or more deployment branches if, for example, you cannot determine the release time yourself, for example if an iOS application has to pass the app store validation or you only have a fixed time window available for deployment. In this case, a production branch `prod`, that reflects the code provided is recommended. Such a workflow prevents the additional work when using *Git rebase* and *Git tags*.

Assuming that you have a `development`, `staging` and `production` environment, then a merge or pull request for a feature is first made to the `staging` branch. As long as the quality check has been passed there, the changes and the code can be ready for production, the changes can be transferred to the `main` branch. This process can be repeated several times for new features until for example the time has come for the *going live* of these changes and a deployment branch can be created.

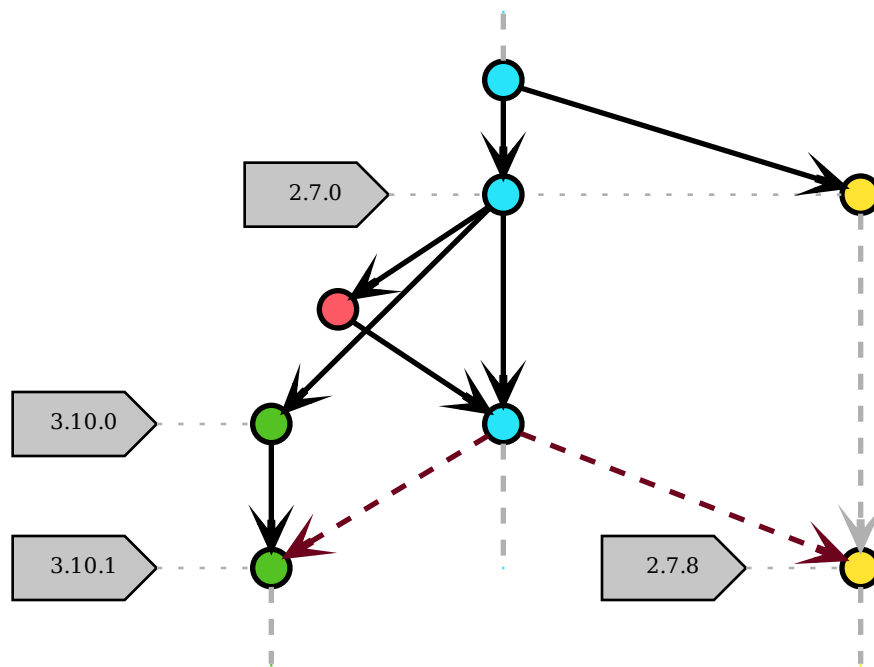


Release branches

Release branches are recommended when software is to be delivered to customers. In this case each branch should contain a minor version, for example 2.7 or 3.4. Usually these branches are created from the `main` branch as late as possible. This reduces the number of merges that have to be distributed across multiple branches during bug fixes. Usually, these are first transferred to the `main` and then transferred from there to the release branch with *Git cherry-pick*, for example:

```
$ git switch 3.10
$ git cherry-pick 61de025
[3.10 b600967] Fix bug #17
Date: Thu Sep 15 11:17:35 2022 +0200
1 file changed, 9 insertions(+)
```

This upstream first approach is for example used by [Google](#) and [Red Hat](#). Every time a bug fix has been adopted in a release branch, the release is increased by a patch version with a [Tag](#), see also [Semantic Versioning](#).



Trunk Based Development

[Trunk Based Development](#) recommends short-lived topic branches that are merged into a single main branch. TBD (Trunk Based Development) leads to an easily managed linear progression.

Trunk Based Development is a perfect fit for one-person projects. Branches are not necessary, but using a version control system pays off quickly even for a single developer.

In smaller development teams, each pair-programming duo preferably transfers small commits directly to the trunk (or main branch), although the build must first be successfully executed before integration.

Trunk based development on a large scale is best done with short-lived feature branches, where one person develops over a few days at most, and the changes are then integrated into the trunk (or main) with pull or merge requests, code review and build automation.

Merge strategies: merge vs. squash vs. rebase

I use `git merge`, `git merge squash` and `git rebase` depending on the situation. They all have their merits, but their use depends very much on the context.

`git merge`

adds a new commit when the branches are merged.

This has the advantage that it best represents the true history. You can see the merge and all the WIP (work in progress) commits that were run during development. If necessary, you can simply undo the merge with `git revert -m/--mainline 1/2 MERGE-COMMIT_SHA`.

-m 1

takes you back to the behaviour of the parent element from the branch to which you have applied the changes.

-m 2

takes you back to the behaviour of the parent element from the branch from which you have applied the changes.

Tip: More commits also make *git bisect* better, as long as a build can be created for each commit. With a hundred or at most a thousand lines that have changed, I still have a chance of finding the bug in a reasonable amount of time.

See also:

- [Advanced Merging](#)

git merge --squash

allows you to merge all changes from a branch into a single commit above the current branch.

This is useful if you have many small WIP commits that are really all aimed at one feature. When squashing, I make sure to rewrite the commit message so that it is as meaningful as possible. The usual squash commit message created by Git, *GitLab* etc. is usually not sufficient and simply adds all squash commit messages together, possibly a series of WIP commit messages.

git rebase

moves a sequence of commits to a new base commit. With `git rebase`, the advantage to find a bug quickly using *git bisect* remains. In addition, however, it is now easier to recognise the context in which the bug occurred.

Tip: With a large diff and many WIP commits, `git rebase` can be used interactively to selectively choose commits for the squash option and rearrange the commits. However, it only does one thing at a time:

- merge commits with the `squash` option or
- change the order of the commits or
- edit the commits.

Do not try to make all changes at once.

Tip: If you don't feel safe with `git rebase`, then don't do it! You can use `git merge` or `git commit --amend` instead.

See also:

- [Git rebase](#)
- [Rewriting History: Squashing Commits](#)
- [Rewriting History: Reordering Commits](#)

Change commits for a clean log

With `git commit --fixup` and `git rebase --autosquash` you can correct a series of commits relatively easily. To demonstrate this with an example, I present the following scenario:

1. We have two commits in our `my-feature` branch: one for the actual function, the other for the associated tests:

```
$ git log --oneline my-feature ^origin/main
a4587fa (my-feature) Add test for my new feature
56e34e9 Add new feature
```

2. During the *merge* or *pull request*, we receive feedback on both our function and our tests, which we would like to integrate into our existing commits. To do this, we first create temporary commits:

```
$ git commit -m "Feedback on the tests from my function"
$ git commit -m "Feedback on my function"
$ git log --oneline my-feature ^origin/main
556c1e8 (my-feature) Feedback on my function
8780db6 Feedback on the tests from my function
a4587fa Add test for my new feature
56e34e9 Add new feature
```

... with `git rebase`

3. With `git rebase -i` we can interactively rearrange the pick lines:

```
$ git rebase -i origin/main
```

This opens our editor:

```
pick 56e34e9 Add new feature
pick a4587fa Add test for my new feature
pick 8780db6 Feedback on the tests from my function
pick 556c1e8 Feedback on my function
```

We can then change the lines, for example to:

```
pick 56e34e9 Add new feature
squash 556c1e8 Feedback on my function
pick a4587fa Add test for my new feature
squash 8780db6 Feedback on the tests from my function
```

Now we have two commits again:

```
$ git log --oneline my-feature ^origin/main
31a140a (my-feature) Add test for my new feature
132ae9b Add new feature
```

4. The changes can now be sent to our remote branch with `git push -f`.

...with `git commit --fixup` and `git rebase --autosquash`

In Git, however, there is an even easier way to correct a previous commit: with `git commit--fixup` and `git rebase --autosquash`.

5. We create two temporary commits again, but this time with `git commit--fixup`:

```
# Further changes to the tests
$ git commit --fixup=31a140a
[my-feature dd0c0d1] fixup! Add test for my new feature
1 file changed, 1 insertion(+)
# Further changes to my function
$ git commit --fixup=132ae9b
[my-function bc2298a] fixup! Add new feature
1 file changed, 1 insertion(+)
$ git log --oneline my-feature ^origin/main
bc2298a (my-feature) fixup! Add new feature
dd0c0d1 fixup! Add test for my new feature
31a140a Add test for my new feature
132ae9b Add new feature
```

For commits with the `--fixup=SHA` option, Git writes a specially formatted commit message that can be read as *this commit corrects that commit*.

6. Instead of using `git rebase -i` to manually specify the Pick/Squash lines, we can now simply run `git rebase --autosquash`:

```
$ git rebase --autosquash origin/main
Successfully rebased and updated refs/heads/my-feature.
$ git log --oneline my-feature ^origin/main
694cb48 (my-feature) Add test for my new feature
55cbe9b Add new feature
```

`git rebase --autosquash` automates what we have just done manually with `git rebase -i` – but it does not open an editor in which we have to move the commits manually.

Tip: The `--fixup` option also contains the `amend` and `reword` options to reformulate the commit message, for example `git commit --fixup:amend=SHA`.

Further options can be found in the [Git commit documentation](#).

Monorepos and large repositories

In a large project, single components of a software may be kept in separate repositories. However, sometimes this creates unnecessary complexity, for instance which versions of the components work together. In these cases, it can make sense to keep all parts of a project in a monolithic repository or *monorepo*.

Definition

- The repository contains more than one logical project (for example an iOS client and a web application).
- The logical projects can be built, tested and deployed independently.
- These projects are usually only loosely connected or can be connected in other ways, for example via dependency management tools.
- The repository contains many commits, branches and/or tags. Or it contains many and/or large files.

With thousands of commits by hundreds of authors in thousands of files per month, the [Linux kernel repository](#) is huge.

Pros and cons

One advantage of monorepos may be that the effort to determine which versions of one project are compatible with which versions of another project may be significantly reduced. This is at least always the case if all projects of a Repository are worked on by only one developer team. Then it is recommended to receive with each Merge again a executable version also if the API between the two projects was changed.

However, performance losses can prove to be a disadvantage. These can arise, for example, from

a large number of commits

Since Git uses DAGs (directed acyclic graphs) to represent the history of a project, all operations that traverse this graph, for example `git log` or `git blame`, will become slow.

a large number of Git references

A large number of branches and tags also slow down git. You can use `git ls-remote` to view the refs in a repository, and `git gc` to combine loose refs into a single file.

Any operation that must traverse the commit history of a repository and account for the individual refs, such as with `git branch --contains *COMMIT`, will be slow on a repo with many refs.

a large number of versioned files

The directory cache index (`.git/index`) is used by Git to determine if the file has been modified. In doing so, as the number of files increases, many operations, such as `git status` and `git commit`, slow down.

large files

Large files in a subtree or project reduce the performance of the entire repository.

Strategies for large repositories

The design goals of Git that have made it so successful and popular sometimes conflict with the desire to use it in ways for which it was not designed. Nevertheless, there are a number of strategies that can be helpful when working with large repositories:

`git clone --depth`

Even though the threshold at which a history is considered huge is quite high, it can still be tedious to clone it. Nevertheless, we cannot always avoid long histories when they need to be maintained for legal or regulatory reasons.

The solution for a fast clone of such a repository is to copy only the most recent revisions. With the shallow option of `git clone` you can retrieve only the last *N* commits of the history, for example `git clone --depth N REMOTE-URL`.

Tip: Build systems connected to your Git repository also benefit from such shallow clones!

Shallow clones have been rather rare in Git until now, as some operations were hardly supported at the beginning. For some time now (in versions 1.9 and higher) you can even perform pull and push operations in repositories from a Shallow Clone.

`git filter-branch`

For large repositories where many binaries have been accidentally transferred, or old assets that are no longer needed, `git filter-branch` is a good solution to go through the entire history and filter out, change or skip files according to predefined patterns.

It's a very powerful tool once you figure out where your repository is heavy. There are also helper scripts to identify large items: `git filter-branch --tree-filter 'rm -rf /PATH/TO/BIG/ASSETS'`

Warning: However, `git filter-branch` rewrites the entire history of your project, that is, on the one hand all commit hashes change and on the other hand every team member has to clone the updated repository again.

See also:

- [How to tear apart a repository: the Git way](#)

`git clone --branch`

You can also limit the size of the cloned history by cloning a single branch, for example with `git clone REMOTE-URL --branch BRANCH-NAME --single-branch FOLDER`.

This can be useful if you are working with long-running and divergent branches, or if you have many branches and only need to work with some of them. However, if you only have a few branches with few differences, you probably won't notice much difference with this.

Git LFS

[Git LFS](#) is an extension that stores pointers to large files in your repository rather than the files themselves; these are stored on a remote server, drastically reducing the time it takes to clone your repository. Git LFS accesses Git's native push, pull, checkout and fetch operations to transfer and replace objects, meaning you can work with large files in your repository as usual.

You can install Git LFS with

```
$ sudo apt install git-lfs
```

```
$ brew install git-lfs
```

Git LFS can be installed with [git for windows](#).

Then you can install Git LFS in your repository with

```
$ git lfs install
Updated Git hooks.
Git LFS initialized.
```

Now, to apply Git LFS to specific file types, you can for example specify:

```
$ git lfs track "*.pdf"
Tracking "*.pdf"
```

This creates the following line in your `.gitattributes` file:

```
*.pdf filter=lfs diff=lfs merge=lfs -text
```

Finally, you should manage the `.gitattributes` file with Git:

```
$ git add .gitattributes
```

git-sizer

`git-sizer` calculates various metrics for a local Git repository and flags those that might cause you problems or inconvenience, for example:

```
$ git-sizer
Processing blobs: 1903
Processing trees: 4126
Processing commits: 1055
Matching commits to trees: 1055
Processing annotated tags: 2
Processing references: 5
```

Name	Value	Level of concern
Biggest objects		
* Blobs		
* Maximum size [1]	35.8 MiB	***

```
[1] 9fe7b8048891965e476aac0410e08e050fd21354 (refs/heads/main:docs/workspace/pandas/
↳ descriptive-statistics.ipynb)
```

Installation

1. Go to the [releases](#) page and download the ZIP file that corresponds to your platform.
2. Unpack the file.
3. Move the executable file (`git-sizer` or `git-sizer.exe`) into your `PATH`.

```
$ brew install git-sizer
```

Git file system monitor (FSMonitor)

`git status` and `git add` are slow because they have to search the entire working tree for changes. The `git fsmonitor--daemon` function, available in Git version 2.36 and later, speeds up these commands by reducing the scope of the search:

```
$ time git status
On branch master
Your branch is up to date with 'origin/master'.
```

(continues on next page)

(continued from previous page)

```

real    0m1,969s
user    0m0,237s
sys     0m1,257s
$ git config core.fsmonitor true
$ git config core.untrackedcache true
$ time git status
On branch master
Your branch is up to date with 'origin/master'.
real    0m0,415s
user    0m0,171s
sys     0m0,675s
$ git fsmonitor--daemon status
fsmonitor-daemon is watching '/srv/jupyter/linux'

```

See also:

- [Improve Git monorepo performance with a file system monitor](#)
- [Scaling monorepo maintenance](#)

Scalar

scalar, a repository management tool for large repositories from [Microsoft](#), has been part of the Git core installation since version 2.38. To use it, you can either clone a new repository with `scalar clone /path/to/repo` or apply scalar to an existing clone with `scalar register /path/to/repo`.

Other options of `scalar clone` are:

-b, --branch BRANCH

Branch to be checked out after cloning.

--full-clone

Create full working directory when cloning.

--single-branch

Download only metadata of the branch that will be checked out.

With `scalar list` you can see which repositories are currently tracked by Scalar and with `scalar unregister /path/to/repo` the repository is removed from this list.

By default, [Sparse-Checkout](#) is enabled and only the files in the root of the git repository are shown. Use `git sparse-checkout set` to expand the set of directories you want to see, or `git sparse-checkout disable` to show all files. If you don't know which directories are available in the repository, you can run `git ls-tree -d --name-only HEAD` to find the directories in the root directory, or `git ls-tree -d --name-only HEAD /path/to/repo` to find the directories in `/path/to/repo`.

See also:

[git ls-tree](#)

To enable sparse-checkout afterwards, run `git sparse-checkout init --cone`. This will initialise your sparse-checkout patterns to match only the files in the root directory.

Currently, in addition to sparse-checkout, the following functions are available for scalar:

- [FSMonitor](#)
- [multi-pack-index \(MIDX\)](#)

- [commit-graph](#)
- [Git maintenance](#)
- Partial cloning with `git clone --depth` and `git filter-branch`

The configuration of `scalar` is updated as new features are introduced into Git. To ensure that you are always using the latest configuration, you should run `scalar reconfigure /PATH/TO/REPO` after a new Git version to update your repository's configuration, or `scalar reconfigure -a` to update all your Scalar-registered repositories at once.

See also:

- [Git - scalar Documentation](#)

Splitting repos

It is often useful to divide a large Git repository into multiple smaller ones. This can be necessary in a project that has grown over time, or if you want to manage a sub-project in a separate repository. Of course you could simply create a new repository and copy the files, but you would also lose the entire version history.

Here I describe how you can split a Git repository without losing the associated history.

Scenario and goals

We want to split out from the Jupyter tutorial repository the part that deals with visualising the data: `docs/viz/`. The challenge is that the history for the `docs/viz/` directory is mixed with other changes. Therefore, we first clone the same repository twice:

```
$ git clone git@github.com:veit/jupyter-tutorial.git
Klone nach 'jupyter-tutorial'...
$ git clone git@github.com:veit/jupyter-tutorial.git pyviz-tutorial
Klone nach 'pyviz-tutorial' ...
```

The next step is to filter out the unwanted histories from each of the two repos. To rewrite the history and keep only those commits that actually affect your content of a particular subfolder, we use [git-filter-repo](#):

```
$ curl https://raw.githubusercontent.com/newren/git-filter-repo/main/git-filter-repo -o_
↪git-filter-repo
% Total    % Received % Xferd    Average Speed   Time    Time       Time    Current
           Dload  Upload   Total     Spent    Left     Speed
100 161k  100 161k    0     0  578k      0 --:--:-- --:--:-- --:--:--  584k

$ cd pyviz-tutorial
$ python3 ../git-filter-repo --path docs/viz
```

The only thing left to do now is to adjust the remote URL:

```
$ git remote add origin git@github.com:veit/pyviz-tutorial.git
$ git push -u origin main
```

For our Jupyter tutorial repository, we now invert the selected path:

```
$ cd jupyter-tutorial
$ python3 ../git-filter-repo --invert-paths --path docs/viz
```

(continues on next page)

(continued from previous page)

```
$ git remote add origin git@github.com:veit/jupyter-tutorial.git  
$ git push -f -u origin main
```

CI-friendly Git Repos

In the following, I'd like to share some tips on how Git repositories and [Continuous Integration](#) can work well together with [GitLab CI/CD](#) or [GitHub Actions](#).

Store large files outside your repository

Every time a new build is created, the working directory needs to be cloned. However, if your repository is bloated with large artefacts, it will slow down and you will have to wait longer for the results.

However, if your build depends on binaries from other projects or large artefacts, it may be useful to have an external storage system that provides those files you need in the build directory at the start of your build for download.

Use shallow clones

Every time a build is executed, your build server clones your repository into the current working directory. Git usually clones the entire history of the repository, so this process takes longer and longer over time. Unless you use so-called shallow clones, where only the current snapshot of the repository is pulled down with `git clone --depth` and only the relevant branch with `git clone --branch`. This shortens the build time, especially for repositories with a long history and many branches.

In doing so, since version 1.9, Git can make simple changes to files, such as updating a version number, without pushing the entire history.

Warning: In a shallow clone, git fetch can result in an almost complete commit history being downloaded. Other git operations can also lead to unexpected results and negate the supposed advantages of shallow clones, so we recommend using shallow clones only for builds and deleting the repository immediately afterwards.

However, if you want to continue using the repositories, the following tip may be helpful.

Cache the repo on build servers

This also speeds up cloning as the repos only need to be updated.

Note: Repo caching is only beneficial if the build environment persists from build to build. However, if your build agent, for example Amazon EC2, dismantles the build again, you have nothing to gain with caching.

Choose triggers wisely

It's a good idea to run CI on all your active branches. But it's not a good idea to run all builds on all branches against all commits.

Typically we give everyone on the development team the option to do their branch builds at the click of a button, rather than triggering them automatically. This seems like a good way for us to balance regular testing with saving resources. However, in critical branches like `main` or `stable`, builds are triggered automatically. In addition, we also get automated timely test results for any merge or pull request.

Typically we give everyone on the development team the option to do their branch builds at the click of a button, rather than triggering them automatically. This seems like a good way for us to balance regular testing with saving resources. However, in critical branches like `main` or `stable`, builds are triggered automatically. In addition, we also get automated timely test results for any merge or pull request.

Advanced Git

git cherry-pick

allows you to append any Git commit to the current HEAD based on its hash value.

git bisect

allows you to quickly find a Git commit that has introduced a regression.

git notes

adds text notes to commits, tags and other objects.

Git hooks

are scripts that are automatically executed when certain events occur in a Git repository.

Jupyter Notebooks

can lead to problems when managing with Git.

Binärdateien

can be configured in Git so that meaningful diffs are displayed.

Visual Studio Code

can use an existing Git installation to provide the corresponding functionalities.

GitLab

is a web application for version management based on Git.

git-big-picture

visualises Git repositories as DAGs (directed acyclic graph).

etckeeper

is a collection of tools that can be used to manage the `/etc` directory in a Git repository.

Git's database internals

refers to articles on Git's database internals.

Git cherry-pick

`git cherry-pick` allows you to append arbitrary Git commits to the current HEAD based on their hash value. Cherry-picking is selecting a commit from one branch and applying it to another, for example:

```
$ git checkout 3.10
$ git cherry-pick 61de025
[3.10 b600967] Fix bug #17
Date: Thu Sep 15 11:17:35 2022 +0200
1 file changed, 9 insertions(+)
```

Thereby `git cherry-pick` can be used with different options:

--edit, -e

does not take over the existing commit message but allows you to create your own commit message for this cherry-pick.

--no-commit, -n

does not create a new commit but moves the contents of the commit to the working directory.

--signoff, -s

adds a signature line with `signed-off-by` at the end of the commit message.

`git cherry-pick` also accepts options to resolve merge conflicts, including `--abort`, `--continue` and `--quit`.

`git cherry-pick` can be useful for reverting changes, for example if a commit was accidentally made to the wrong branch, you can switch to the branch where the change was supposed to be made and then cherry-pick the commit to that branch.

However, cherry-picking usually results in duplicate commits, and in many cases we prefer to use git merges. Nevertheless, `git cherry-pick` can be very suitable for some scenarios, for example [Release branches](#) workflows.

git range-diff

`git range-diff` shows the difference between two commit ranges, that is, which commits between these ranges are the same or have changed. This command can help, for example, when checking which commits were applied to which branches with `git cherry-pick`.

Find regressions with git bisect

`git bisect` allows you to quickly find a git commit that has introduced a regression. The name *bisect* comes from the [binary search](#) that the command uses. The list of commits is repeatedly halved until the relevant commit is found. This means that only $\log_2(n+1)$ commits need to be tested.

1. To do this, start the search with `git bisect start`. You can then use `git bisect new [COMMIT]` and `git bisect old [COMMIT]` to narrow down the area in which an error was introduced. Alternatively, the short form `git bisect start [BAD COMMIT] [GOOD COMMIT]` can also be used. `git bisect` then checks out a commit in the middle and asks you to test it, for example:

```
$ git bisect start v2.6.27 v2.6.25
Bisecting: 10928 revisions left to test after this (roughly 14 steps)
[2ec65f8b89ea003c27ff7723525a2ee335a2b393] x86: clean up using max_low_pfn on 32-bit
```

2. The search can now be continued manually or automatically with a script. Manually, you can use `git bisect new` and `git bisect old` to narrow down the area in which an error was introduced. If this commit is found, the output may look like this, for example:

```
$ git bisect new
2ddcca36c8bcfa251724fe342c8327451988be0d is the first bad commit
commit 2ddcca36c8bcfa251724fe342c8327451988be0d
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date: Sat May 3 11:59:44 2008 -0700

    Linux 2.6.26-rc1

:100644 100644 5cf82581... 4492984e... M      Makefile
```

3. We then use `git show HEAD` to check what changes have been made in this commit:

```
$ git show HEAD
commit 2ddcca36c8bcfa251724fe342c8327451988be0d
Autor: Linus Torvalds <torvalds@linux-foundation.org>
Datum: Sa 3. Mai 11:59:44 2008 -0700

    Linux 2.6.26-rc1

diff --git a / Makefile b / Makefile
index 5cf8258 ..4492984 100644
--- a / Makefile
+++ b / Makefile
@@ -1,7 +1,7 @@
VERSION = 2
PATCHLEVEL = 6
-SUBLEVEL = 25
-EXTRAVERSION =
+ SUBLEVEL = 26
+ EXTRAVERSION = -rc1
NAME = Funky Weasel ist Jiggy wit it

# * DOKUMENTATION *
```

4. Finally, you can use `git bisect reset` to return to the branch you were in before the bisect search:

```
$ git bisect reset
Checking out files: 100% (21549/21549), done.
Previous HEAD position was 2ddcca3... Linux 2.6.26-rc1
Switched to branch 'master'
```

Mark non-testable commits with `git bisect skip`

Sometimes with `git bisect` you end up with a commit that you can't test because there's another problem. Usually this is due to an error that prevents you from running your code or seeing the test result, for example a syntax error. In this case, you should not mark the commit as old or new, as you will not be able to determine the behaviour due to the error. Instead, you should skip the commit with `git bisect skip`. `git bisect` checks out a neighbouring commit for testing instead. If this works, continue testing and executing new or old as usual. If not, run `git bisect skip` again. If you know that there is a range of untestable commits, instruct `git bisect` to skip this entire area with `git bisect skip COMMIT1..COMMIT2`.

See also:

- Avoiding testing a commit

Automatic testing with `git bisect run`

It is often possible to automate the test of whether a commit shows old or new behaviour. This speeds up the use of `git bisect` massively, as you no longer have to make an entry at every step. It also makes the process less error-prone, as you won't accidentally execute the wrong old and new subcommand. Automated tests are also advantageous if your test process takes a while, for example if you have a long compilation step. The search will not be interrupted to wait for your input, and you can work on something else in the meantime.

To start automatic tests, use `git bisect run` with your test command and optional arguments. You may need to create a short test script that runs the affected part of your code and checks what behaviour is present. `git bisect` runs the specified command at each step of the binary search loop and uses its results to call `old`, `new` or `skip` as needed.

You can find an example of this in the issue [fetch_california_housing fails in CI on master](#) from scikit-learn:

```
$ git bisect run pytest sklearn/utils/tests/test_multiclass.py -k test_unique_labels_non_
↪specific
```

Automated testing of performance regressions

With a little extra effort, you can use automated tests to search for more complicated changes in behaviour. For performance tests, we need a test programme that can perform multiple runs and determine the minimum time while eliminating possible noise:

```
from subprocess import run
from time import perf_counter

times = []
for _ in range(10):
    start = perf_counter()
    run(
        ['./perftest', PARAM],
        check=True,
        capture_output=True,
    )
    elapsed = perf_counter() - start
    times.append(elapsed)
if min(times) > X.0:
    print("Too slow")
    raise SystemExit(1)
else:
    print("Fast enough")
    raise SystemExit(0)
```

The programme executes `python perftest.py PARAM` ten times and measures the time for each execution. It then compares the minimum execution time with a limit value of `X` seconds. If the minimum time is above the limit value, it outputs *Too slow* and exits with the exit code 1, otherwise it outputs *Fast enough* and exits with the exit code 0:

```
$ python perftest.py PARAM
Fast enough
$ echo $? 0
```

Reproducing the binary search with `git bisect log` and `git bisect replay`

The scikit-learn issue also shows how you can communicate the results of your bisect search to others in a reproducible way using `git bisect log`:

```
$ git bisect log
81f2d3a0e *    massich/multiclass_type_of_target Merge branch 'master' into multiclass_
↪type_of_target
    |\
15f24f25d | * bad DOC Cleaning for what's new
fbb2c7c70 | * good-fbb2c7c7007dc373c462e39ab273a183a8823d58 @ ENH Adds _
↪MultimetricScorer for Optimized Scoring  (#14593)
...
```

With `git bisect log > bisect_log.txt` you can also save your search for others to reproduce:

```
$ git bisect replay bisect_log.txt
```

Git Notes

[Git Notes](#) add text notes to commits, tags and other objects. Such notes can contain all kinds of metadata, for example comments on code review, links to bug reports, etc:

1. Add a git note:

```
$ git notes add -m 'Example note'
```

2. Display a git note:

```
$ git log
commit 859de540cda23f510f4ecbe0f38d07666e933f08 (HEAD -> main)
Author: Veit Schiele <veit@cusy.io>
Date:   Sun Mar 24 11:17:56 2024 +0100

    A commit message

Notes:
    Example note
```

3. Change a git note:

```
$ git notes edit
```

However, Git notes are not sent to the remote repository with `git push` or `git pull` by default; they must be synchronised with `git push origin 'refs/notes/*'` and `git fetch origin 'refs/notes/*:refs/notes/*'`.

Warning: Do not use `git pull` instead of `git fetch`: you will not be able to merge `refs/notes/commits` with your current branch.

Note: Git notes are not included in the git commit history, so they cannot be used for regulatory purposes where provenance, non-repudiation or tamper resistance must be proven. However, they can be useful for build tags and

similar.

See also:

- [Git Notes: Git's Coolest, Most Unloved Feature](#)
- [git-appraise](#)
- [github-issues-git-notes](#)

Git hooks

Git hooks are scripts that are automatically executed when certain events occur in a Git repository, including:

Command	Hook
commit	commit-msg, pre-commit
merge	pre-merge, commit-msg
rebase	pre-rebase
pull	pre-merge, commit-msg
push	pre-push

They can be located either in local or server-side repositories. This allows Git repositories to be customised and user-defined actions to be triggered.

Git hooks are located in the `.git/hooks/` directory. When a repository is created, some sample scripts are already created there:

```
.git/hooks/
├── applypatch-msg.sample
├── commit-msg.sample
├── fsmonitor-watchman.sample
├── post-update.sample
├── pre-applypatch.sample
├── pre-commit.sample
├── pre-merge-commit.sample
├── prepare-commit-msg.sample
├── pre-push.sample
├── pre-rebase.sample
├── pre-receive.sample
└── update.sample
```

For the scripts to be executed, only the suffix `.sample` must be removed and, if necessary, the file permission must be executable, for example with `chmod +x .git/prepare-commit-msg`.

The integrated scripts are shell and Perl scripts, but any scripting language can be used. The Shebang line (`#!/bin/sh`) determines how the file is to be interpreted.

However, the scripts cannot be copied into the server-side repository.

pre-commit framework

`pre-commit` is a framework for managing and maintaining multilingual commit hooks.

An essential task is to make the same scripts available to the entire development team. `pre-commit` by yelp manages such hooks and distributes them to different projects and developers.

Git hooks are mostly used to automatically point out problems in the code before code reviews, for example to check the formatting or to find debug statements. `pre-commit` simplifies the sharing of hooks across projects. The language in which a linter was written, for example, is abstracted away – `scss-lint` is written in Ruby, but you can use it with `pre-commit` without having to add a Gemfile to your project.

Installation

Before you can execute the hooks, the `pre-commit` framework must be installed:

Before the `pre-commit` framework can be installed with Pipenv, the [Microsoft Build Tools for C++](#) must be downloaded and executed so that the *Desktop development with C++* can be selected and installed with the standard options.

Only then can the `pre-commit` framework be installed with:

```
$ pipenv install pre-commit
```

```
$ apt install pre-commit
```

```
$ brew install pre-commit
```

```
$ pipenv install pre-commit
```

Check the installation for example with

```
$ pipenv run pre-commit -V
pre-commit 2.21.0
```

Configuration

After Pre-Commit is installed, the `.pre-commit-config.yaml` file in the root directory of your project can be used to configure plugins for this project.

```
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v3.2.0
  hooks:
  - id: trailing-whitespace
  - id: end-of-file-fixer
  - id: check-yaml
  - id: check-added-large-files
```

You can also generate such an initial `.pre-commit-config.yaml` file with

```
$ pipenv run pre-commit sample-config > .pre-commit-config.yaml
```


If you want to apply `check-json` to your Jupyter notebooks, you must first configure that the check should also be used for the file suffix `.ipynb`:

```
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v3.2.0
  hooks:
  ...
- id: check-json
  types: [file]
  files: \.(json|ipynb)$
```

See also:

For a full list of configuration options, see [Adding pre-commit plugins to your project](#).

You can also write your own hooks, see [Creating new hooks](#).

Installing the git hook scripts

To ensure that pre-commit is also reliably executed before each commit, the script is installed in our project:

```
$ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

If you want to uninstall the git hook scripts, you can do so with `pre-commit uninstall`.

Run

```
pre-commit run --all-files
```

runs all pre-commit hooks independently of `git commit`:

```
$ pipenv run pre-commit run --all-files
Trim Trailing Whitespace.....Passed
Fix End of Files.....Passed
Check Yaml.....Passed
Check for added large files.....Passed
```

`pre-commit run HOOK`

executes single pre-commit hooks, for example `pre-commit run trailing-whitespace`

Note: When a pre-commit hook is called for the first time, it is first downloaded and then installed. This may take some time, for example if a copy of `node` has to be created.

`pre-commit autoupdate`

updates the hooks automatically:

See also:

- [pre-commit autoupdate \[options\]](#).

However, the hooks managed by the pre-commit framework are not limited to being executed before commits; they can also be used for other Git hooks, see [Other pre-commit hooks](#).

pre-commit scripts

pre-commit-hooks

The pre-commit framework already comes with a whole range of scripts, including:

check-added-large-files

prevents large files from being transferred

check-case-conflict

looks for files that would conflict in case-insensitive file systems

check-executables-have-shebangs

makes sure that (non-binary) executables have a shebang line

check-shebang-scripts-are-executable

makes sure (non-binary) files are executable with a shebang line

check-merge-conflict

searches for files containing merge-conflict strings

check-symlinks

checks for symlinks that don't point to anything

destroyed-symlinks

detects symlinks that have been changed into regular files with the contents of the path the symlink points to.

no-commit-to-branch

protects branches before committing

pygrep-hooks

provides regular expressions for Python and reStructuredText, including:

python-no-log-warn

search for the deprecated `.warn()` method of Python loggers

python-use-type-annotations

forces type-annotations to be used instead of type-comments

rst-backticks

detects the use of single backticks when writing reStructuredText

rst-directive-colons

detects that reStructuredText directives do not end with a colon or a space before the colon

rst-inline-touching-normal

detects that inline code is used in normal text in reStructuredText

text-unicode-replacement-char

prevents files that contain UTF-8 Unicode Replacement Characters

Linters and formatters

They are provided in separate repositories, including:

autopep8

provides `autopep8` for the pre-commit framework

mypy

provides `mypy`

validate-pyproject

checks `pyproject.toml` files

sp-repo-review

evaluates existing repos against the [Scientific Python guidelines](#).

clang-format

provides [clang-format-wheel](#)

csslint

provides [csslint](#)

scss-lint

provides [scss-lint](#)

eslint

provides [eslint](#)

fixmyjs

provides [fixmyjs](#)

prettier

provides [prettier](#)

black

for formatting Python code

black

Python code formatter

black-jupyter

Python code formatter for Jupyter notebooks

Python Code Quality Authority

Code quality tools (and plugins) for the Python programming language:

flake8

promotes the enforcement of a consistent Python style

autoflake

removes unused imports and unused variables from Python code

bandit

tool for finding security vulnerabilities in Python code

pydocstyle

static analysis tool to check compliance with Python docstring conventions

docformatter

formats docstrings according to [PEP 257](#)

pylint

Python linter

doc8

executes doc8 for linting documents

prospector

analyses Python code with prospector

isort

sorts Python imports

nbQA

runs isort, pyupgrade, mypy, pylint, flake8 and more on Jupyter notebooks:

nbqa

runs any standard Python code quality tool on a Jupyter notebook

nbqa-black

runs black on a Jupyter notebook

nbqa-check-ast

runs check-ast on a Jupyter notebook

nbqa-flake8

runs flake8 on a Jupyter notebook

nbqa-isort

runs isort on a Jupyter notebook

nbqa-mypy

runs mypy on a Jupyter notebook

nbqa-pylint

runs pylint on a Jupyter notebook

nbqa-pyupgrade

runs ppyupgrade on a Jupyter notebook

nbqa-yapf

runs yapf on a Jupyter notebook

nbqa-autopep8

runs autopep8 on a Jupyter notebook

nbqa-pydocstyle

runs pydocstyle on a Jupyter notebook

nbqa-ruff

runs ruff on a Jupyter notebook

blacken-docs

applies black to Python code blocks in documentation files

Miscellaneous

pyupgrade

automatically updates the syntax for newer versions

reorder-python-imports

reorders imports into Python files

dead

detects dead Python code

python-safety-dependencies-check

analyses Python requirements for known security vulnerabilities

gitlint

Git commit message linter

nbstripout

removes the output of Jupyter Notebooks

ripsecrets

prevents secret keys from being included in your source code

detect-secrets

detects high entropy strings that are likely to be passwords

pip-compile

automatically compiles requirements

kontrollilo

Tool to control licences for OSS dependencies

See also:

- [Supported hooks](#)

Other pre-commit hooks

The hooks managed by the pre-commit framework are not limited to being executed before commits; they can also be used for other Git hooks:

post-commit

As of version 2.4.0, the framework can also execute [post-commit](#) hooks with:

```
$ pipenv run pre-commit install --hook-type post-commit
pre-commit installed at .git/hooks/post-commit
```

However, since `post-commit` does not work on files, all these hooks must set `always_run`:

```
- repo: local
  hooks:
  - id: post-commit-local
    name: post commit
    always_run: true
    stages: [post-commit]
    # ...
```

pre-merge

As of Git 2.24, there is a [pre-merge-commit](#) hook that is triggered after a merge is successful but before the merge commit is created. You can use it with the pre-commit framework with:

```
$ pre-commit install --hook-type pre-merge-commit
pre-commit installed at .git/hooks/pre-merge-commit
```

post-merge

As of version 2.11.0, the framework can also execute scripts for the [post-merge](#) hook:

```
$ pipenv run pre-commit install --hook-type post-merge
pre-commit installed at .git/hooks/post-merge
```

With `$PRE_COMMIT_IS_SQUASH_MERGE` you can find out if it was a squash merge.

pre-push

To use the [pre-push](#) hook with the pre-commit framework, enter the following:

```
$ pre-commit install --hook-type pre-push
pre-commit installed at .git/hooks/pre-push
```

The following environment variables are provided for this purpose:

\$PRE_COMMIT_FROM_REF

The remote revision that was pushed to.

\$PRE_COMMIT_TO_REF

The local revision that was pushed to the remote revision.

\$PRE_COMMIT_REMOTE_NAME

The local revision that was pushed to the remote revision, for example `origin`.

\$PRE_COMMIT_REMOTE_URL

The URL of the remote repository that was pushed to, for example `git@github.com:veit/python4datascience`

\$PRE_COMMIT_REMOTE_BRANCH

The name of the remote branch that was pushed to, for example `refs/heads/TARGET_BRANCH`.

\$PRE_COMMIT_LOCAL_BRANCH

The name of the local branch that was pushed to the remote branch, for example `HEAD`.

commit-msg

`commit-msg` can be used with:

```
$ pre-commit install --hook-type commit-msg
pre-commit installed at .git/hooks/commit-msg
```

The `commit-msg` hook can be configured with stages: `[commit-msg]`, passing the name of a file containing the current contents of the commit message that can be checked.

prepare-commit-msg

`prepare-commit-msg` can be used with `pre-commit` with:

```
$ pre-commit install --hook-type prepare-commit-msg
pre-commit installed at .git/hooks/prepare-commit-msg
```

The `prepare-commit-msg` hook is configured with stages: `[prepare-commit-msg]`, passing the name of a file that contains the initial commit message, for example from `git commit -m "COMMIT-MESSAGE"` to create a dynamic template from it that is displayed in the editor. Finally, the hook should check that no editor is started with `GIT_EDITOR=:`.

post-checkout

The `post-checkout` hook is called when `git checkout` or `git switch` is executed.

The `post-checkout` hook can be used for example for

- checking repositories
- viewing differences from the previous `HEAD`
- changing the metadata of the working directory.

In `pre-commit` it can be used with:

```
$ pre-commit install --hook-type post-checkout
pre-commit installed at .git/hooks/post-checkout
```

Since `post-checkout` does not act on files, `always_run` must be set for all `post-checkout` scripts, for example:

```
- repo: local
  hooks:
  - id: post-checkout-local
    name: Post checkout
    always_run: true
```

(continues on next page)

(continued from previous page)

```
stages: [post-checkout]
# ...
```

There are three environment variables that correspond to the three arguments of `post-checkout`:

\$PRE_COMMIT_FROM_REF

returns the reference of the previous HEAD

\$PRE_COMMIT_TO_REF

returns the reference of the new HEAD, which may or may not have changed.

\$PRE_COMMIT_CHECKOUT_TYPE

returns `Flag=1` if it was a branch checkout and `Flag=0` if it was a file checkout.

post-rewrite

`post-rewrite` is called when commits are rewritten, for example from `git commit --amend` or from `git rebase`.

```
$ pre-commit install --hook-type post-rewrite
pre-commit installed at .git/hooks/post-rewrite
```

Since `post-rewrite` does not affect files, `always_run: true` must be set.

Git tells the `post-rewrite` hook which command triggered the rewrite. `pre-commit` outputs this as `$PRE_COMMIT_REWRITE_COMMAND`.

pre-commit in CI pipelines

Pre-commit can also be used for CI (continuous integration).

Examples for GitHub Actions

pre-commit ci

Service that adds the *pre-commit ci* app to your GitHub repository at <https://github.com/PROFILE/REPOSITORY/installations>.

Besides automatically changing pull requests, the app also [autoupdate](#) to keep your configuration up to date.

You can add further installations under [Install pre-commit ci](#).

.github/workflows/pre-commit.yml

Alternative configuration as a GitHub workflow, for example:

```
name: pre-commit

on:
  pull_request:
  push:
    branches: [main]

jobs:
  pre-commit:
    runs-on: ubuntu-latest
    steps:
```

(continues on next page)

(continued from previous page)

```

- uses: actions/checkout@v3
- uses: actions/setup-python@v3
- uses: actions/cache@v3
  with:
    path: ~/.cache/pre-commit
    key: pre-commit|${{ env.pythonLocation }}|${{ hashFiles('.pre-commit-config.
→yaml') }}
- uses: pre-commit/action@v3.0.1

```

See also:

- [pre-commit/action](#)

Example for GitLab Actions

```

stages:
  - validate

pre-commit:
  stage: validate
  image:
    name: python:3.12
  variables:
    PRE_COMMIT_HOME: ${CI_PROJECT_DIR}/.cache/pre-commit
  only:
    refs:
      - merge_requests
      - tags
      - main
  cache:
    paths:
      - ${PRE_COMMIT_HOME}
  before_script:
    - pip install pre-commit
  script:
    - pre-commit run --all-files

```

See also:

For more information on fine-tuning caching, see [Good caching practices](#).

Skip hooks

Most Git *Git hooks* can be bypassed with the `--no-verify` option. For example, you can skip the pre-commit hook with:

```
$ git commit --no-verify -m "Quick and dirty"
```

If you only want to skip certain *pre-commit scripts*, you can use the environment variable `SKIP` with a comma-separated list of hook IDs, for example:


```
$ SKIP=check-added-large-file,no-commit-to-branch git commit -m "Hotfix"
```

Template for Git repositories

`pre-commit init-templatedir` can be used to set up a template for Git's `init.templateDir` option, whereby any newly cloned repository will automatically receive the pre-commit hooks without having to run `pre-commit install`, for example:

```
$ git config --global init.templateDir ~/.config/git/template
$ pre-commit init-templatedir ~/.config/git/template
pre-commit installed at /Users/veit/.config/git/template/hooks/pre-commit
```

Jupyter Notebooks with Git

Problems with version control of Jupyter Notebooks

There are several issues to manage Jupyter Notebooks with Git:

- Jupyter Notebooks cell metadata changes even when no content changes have been made to the cells. This makes Git diffs unnecessarily complicated.
- The lines that Git writes to the `*.ipynb` files in case of *merge conflicts* cause the notebooks to no longer be valid JSON and therefore cannot be opened by Jupyter: you will then get the *Error loading notebook* message when opening them.

Conflicts are especially common in notebooks because Jupyter changes the following each time a notebook is run:

- Each cell contains a number that indicates the order in which it was executed. If team members execute the cells in different order, every single cell has a conflict! To fix this manually would take a very long time.
- For each image, such as a plot, Jupyter records not only the image itself in the notebook, but also a simple text description containing the ID of the object, for example `<matplotlib.axes._subplots.AxesSubplot at 0x7fbc113dbe90>`. This will change every time you run a notebook, and therefore will conflict every time two people run that cell.
- Some output can be non-deterministic, such as a notebook that uses random numbers or interacts with a service that provides different output over time.
- Jupyter adds metadata to the notebook that describes the environment in which it was last run, such as the name of the kernel. This often varies between different installations, and so two people saving a notebook (even without other changes) will often have a conflict in the metadata.

nbdev2

`nbdev2` has a set of git hooks that provide clean git diffs that automatically resolve most git conflicts and ensure that any remaining conflicts can be fully resolved within the standard Jupyter notebook environment:

- A new git merge driver provides notebook-native conflict markers that result in notebooks opening directly in Jupyter, even if there are Git conflicts. Local and remote changes are each shown as separate cells in the notebook, so you can simply delete the version you don't want to keep or combine the two cells as needed.

See also:[nbdev.merge docs](#)

- Resolving git merges locally is extremely helpful, but we also need to resolve them remotely. For example, if a *merge request* is submitted and then someone else submits the same notebook before the merge request is merged, it could cause a conflict:

```
"outputs": [  
  {  
    <<<<<< HEAD  
      "execution_count": 8,  
    =====  
      "execution_count": 5,  
    >>>>>> 83e94d58314ea43ccd136e6d53b8989ccf9aab1b  
      "metadata": {},
```

The *save hook* of nbdev2 automatically removes all unnecessary metadata (including `execution_count`) and non-deterministic cell output; this means that there are no pointless conflicts like the one above, since this information is not stored in the commits in the first place.

To get started, follow the instructions in [Git-Friendly Jupyter](#).

jq

The results of the calculations can also be saved in the notebook file format [nbformat](#). These can also be Base-64-coded blobs for images and other binary data that should not normally be included in a version management. These can be removed manually with *Cell* → *All Output* → *Clear*, but you have to carry out these steps before every `git add`, and it also does not solve a second cause of the noise in `git diff`, namely some in the [metadata](#).

In order to get systematically comparable versions of notebooks in the version management, we can use [jq](#), a lightweight JSON processor. It takes some time to set up `jq` because it has its own query/filter language, but the default settings are usually well chosen.

Installation

`jq` can be installed with:

```
$ sudo apt install jq
```

```
$ brew install jq
```

Example

A typical call is:

```
jq --indent 1 \  
'(.cells [] | select (has ("output"))) | .outputs) = []  
| (.cells [] | select (has ("execution_count"))) | .execution_count = null  
| .metadata = {"language_info": {"name": "python", "pygments_lexer": "ipython3"}}  
| .Cells []. metadata = {}  
' example.ipynb
```

Each line within the single quotation marks defines a filter – the first selects all entries from the cells list and deletes the output. The next entry resets all outputs. The third step deletes the notebook’s metadata and replaces it with a minimum of necessary information so that the notebook can still be run without complaints. The fourth filter line `.cells [] .metadata = {}`, deletes all meta information. If you want to keep certain meta information, you can indicate this here.

Set up

1. To make your work easier, you can create an alias in the `~/ .bashrc` file:

```
alias nbstrip_jq="jq --indent 1 \
'(.cells[] | select(has(\"outputs\")) | .outputs) = [] \
| (.cells[] | select(has(\"execution_count\")) | .execution_count) = null \
| .metadata = {\"language_info\": {\"name\": \"python\", \"pygments_lexer\": \
↪\"ipython3\"}} \
| .cells[].metadata = {} \
,\""
```

2. Then you can conveniently enter the following in the terminal:

```
$ nbstrip_jq example.ipynb > stripped.ipynb
```

3. If you start with an existing notebook, you should first add a filter commit by simply reading in the newly filtered version of your notebook without the unwanted metadata. After you have added the notebook with `git add`, you can see whether the filter has really worked with `git diff --cached` before you do `git commit -m 'filter'`.
4. If you want to use this filter for all Git repositories, you can also configure your Git globally:

1. First you add the following to your `~/ .gitconfig` file:

```
[core]
attributesfile = ~/.gitattributes

[filter "nbstrip_jq"]
clean = "jq --indent 1 \
'(.cells[] | select(has(\"outputs\")) | .outputs) = [] \
| (.cells[] | select(has(\"execution_count\")) | .execution_count) =
↪null \
| .metadata = {\"language_info\": {\"name\": \"python\", \"pygments_
↪lexer\": \"ipython3\"}} \
| .cells[].metadata = {} \
,\""
smudge = cat
required = true
```

clean

is applied when adding changes to the stage area.

smudge

is used when resetting the workspace by changes from the stage area.

2. Then you have to specify the following in the `~/ .gitattributes` file:

```
*.ipynb filter=nbstrip_jq
```

5. If you then use `git add` to add your notebook to the stage area, the `nbstrip_jq` filter will be applied.

Note: However, `git diff` will not show you any changes between the working and stage areas. Only with `git diff --staged` you can see that only the filtered changes have been applied.

Warning: `clean` and `smudge` filters often do not play well with `git rebase` across such filtered commits. Then you should disable these filters before rebasing.

6. And there is another problem: If such a notebook is run again, `git diff` will not show any changes, but `git status` will. Therefore, the following should be entered in the `~/.bashrc` file to be able to quickly clean the respective working directory:

```
function nbstrip_all_cwd {
  for nbfile in *.ipynb; do
    echo "$( nbstrip_jq $nbfile )" > $nbfile
  done
  unset nbfile
}
```

ReviewNB

[ReviewNB](#) solves the problem of doing [Merge requests](#) with notebooks. GitLab's code review GUI only works with line-based file formats, such as Python scripts. Most of the time, however, I prefer to check the source code notebooks because:

- I want to check the documentation and the tests, not just the implementation
- I want to see the changes to the cell output, like charts and tables, not just the code.

For this purpose [ReviewNB](#) is perfect.

[nbdime](#)

[nbdime](#) is a GUI for [nbformat](#) diffs and replaces [nbdiff](#). It attempts content-aware diffing locally as well as merging notebooks, is not limited to displaying diffs, but also prevents unnecessary changes from being checked in. However, it is not compatible with `nbdev2`.

[nbstripout](#)

[nbstripout](#) automates *Clear all outputs*. It uses [nbformat](#) and a few auto magic to set up `.git config`. In my opinion, however, it has two drawbacks:

- it is limited to the problematic metadata section
- it is slow.

Git for binary files

`git diff` can be configured so that it can also display meaningful diffs for binary files.

... for Excel files

For this we need `openpyxl` and `pandas`:

```
$ pipenv install openpyxl pandas
```

Then we can use `pandas.DataFrame.to_csv` in `exceltocsv.py` to convert the Excel files:

Listing 1: `exceltocsv.py`

```
# SPDX-FileCopyrightText: 2023 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

import sys
from io import StringIO

import pandas as pd

for sheet_name in pd.ExcelFile(sys.argv[1]).sheet_names:
    output = StringIO()
    print("Sheet: %s" % sheet_name)
    pd.read_excel(sys.argv[1], sheet_name=sheet_name).to_csv(
        output, header=True, index=False
    )
    print(output.getvalue())
```

Now add the following section to your global Git configuration `~/.gitconfig`:

```
[diff "excel"]
    textconv=python3 /PATH/TO/exceltocsv.py
    binary=true
```

Finally, in the global `~/.gitattributes` file, our excel converter is linked to `*.xlsx` files:

```
*.xlsx diff=excel
```

... for PDF files

For this, `pdftohtml` is additionally required. It can be installed with

```
$ sudo apt install poppler-utils
```

```
$ brew install pdftohtml
```

Add the following section to the global Git configuration `~/.gitconfig`:

```
[diff "pdf"]
    textconv=pdftohtml -stdout
```

Finally, in the global `~/.gitattributes` file, our pdf converter is linked to `*.pdf` files:

```
*.pdf diff=pdf
```

Now, when `git diff` is called, the PDF files are first converted and then a diff is performed over the outputs of the converter.

... for Word documents

Differences in Word documents can also be displayed. For this purpose [Pandoc](#) can be used, which can be easily installed with

```
$ sudo apt install pandoc
```

```
$ brew install pandoc
```

Download and install the `*.msi` file from [GitHub](#).

Then add the following section to your global Git configuration `~/.gitconfig`:

```
[diff "word"]
    textconv=pandoc --to=markdown
    binary=true
    prompt=false
```

Finally, in the global `~/.gitattributes` file, our word converter is linked to `*.docx` files:

```
*.docx diff=word
```

The same procedure can be used to obtain useful diffs from other binaries, for example `*.zip`, `*.jar` and other archives with `unzip` or for changes in the meta information of images with `exiv2`. There are also conversion tools for converting `*.odt`, `*.doc` and other document formats into plain text. For binary files for which there is no converter, strings are often sufficient.

Visual Studio Code

Visual Studio Code can use an existing [Git installation](#) to provide the corresponding functionalities.

Clone

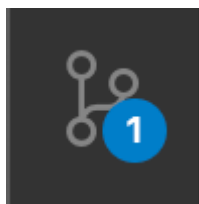


Fig. 1: Source control icon

450 If you have not yet opened a repository, you have the option of selecting *Open*

Gutter indicators

When you open a Git repository and start making changes, VS Code adds useful annotations:

- a red triangle indicates where lines have been deleted
- a green bar indicates newly added lines
- a blue bar indicates lines that have been changed.

Commit

`git add` and `git reset` can be selected either in the context menu of a file or by drag & drop. After a `git commit`, you can enter a commit message and confirm with `Ctrl` or `.`. If there are already changes in the stage area, only these will be committed; otherwise you will be asked to select changes. If necessary, you will receive more specific commit actions in *Views and More Actions...*

Note: If you have accidentally created your commit in the wrong branch, you can undo it with *Git: Undo Last Commit* in the *Command Palette* (`Ctrl` + `P`).

The source control icon in the activity bar on the left shows you how many changes you have made in your repository. Selecting the icon will give you a more detailed overview of your changes. Selecting a single file will show you the line-by-line text changes. You can also use the editor on the right to make further changes.

Branches and tags

You can create branches and switch to them using *Git: Create Branch* and *Git: Checkout to* from the *Command Palette* (`Ctrl` + `P`). When you call *Git: Checkout to*, a dropdown list appears with all the branches and tags of the repository. You can also create a new branch here.

Git status bar



Fig. 2: Status bar

In the lower left corner you will see the status display with further indicators about the state of your repository:

- the current branch with the possibility to switch to another branch
- incoming and outgoing commits
- the *Synchronize Changes* action, which first executes `git pull` and then `git push`.
- [Git Blame](#)
- [Git History](#)
- [Git Lens](#)
- [GitLab VS Code Extension](#)

GitLab VS Code Extension

[GitLab VS Code Extension](#) integrates GitLab 13.0 into Visual Studio Code:

Display GitLab issues and merge requests

Issues, comments, merge requests and changed files are displayed in the sidebar or in a [custom search](#).

Create and review merge requests

Issues can be commented directly in VS Code, and [GitLab Slash Commands](#) are also supported. You can create, edit and delete comments in the diff view of a merge request.

Configuring and validating GitLab CI/CD

You can edit the `gitlab-ci.yml` file, which automatically completes the variables. You can also validate the file locally.

Search the repository without cloning

You can access repositories read-only, provided that an access token is registered for the corresponding GitLab instance.

GitLab

[GitLab](#) is a web application for version management based on Git. Later, further functions were added such as an issue tracking system with Kanban board, a system for [Continuous Integration and Continuous Delivery \(CI/CD\)](#) as well as a Wiki and Snippets. The GitLab Community Edition (CE) is developed as open source software under the MIT licence and can be installed on-premises.

The GitLab CI tools enable automated builds and deployments without the need for external integrations. If a PaaS solution such as [Kubernetes](#) is already in use, apps can be automatically deployed, tested and scaled with GitLab CI/CD. In addition, code can be automatically scanned for potential security risks.

GitLab is a completely packaged platform, while GitHub can be extended with apps from the Marketplace. However, this does not mean that GitLab cannot be integrated, for example with Asana, Jira, Microsoft Teams, Slack, etc.

See also:

Visual Studio Code: [GitLab Workflow](#)

Roles, groups and permissions

Depending on the role a person has in a particular group or project, they have different permissions. If the person is in both a project group and the project, the highest role is used.

See also:

- [Permissions and roles](#)

Members of a project

Members are the people and groups who have access to your project. Each member is given a role that determines what they can do in the project. Project members can:

- be direct members of the project.
- inherit membership of the project from the project group.
- be a member of a group shared with the project.
- be a member of a group shared with the project group.

Permissions in GitLab

Guests

are not active contributors to private projects; they can only see and leave comments and issues.

Reporters

participate as readers. They cannot write to the repository, but they can contribute to issues.

Developers

contribute directly and have access to everything from idea to production, unless something has been explicitly restricted, for example by branch protection.

Maintainer

can push to `main` and move code into the `production` environment.

Owners

essentially administer the groups and workflows. They can grant access to groups and are allowed to delete.

Protected branches

In GitLab, permissions are basically defined to give read or write permissions to the repository and branches. To impose further restrictions on certain branches, they can be protected. The default branch for your repository is protected by default. When a branch is protected, the following restrictions are usually enforced on the branch by default:

Action	Role
Protect a branch	Maintainer
Push into this branch	GitLab admins and anyone explicitly allowed to do so.
Force push into this branch	No one
Delete the branch	With a Git command, nobody; with GitLab UI or API, at least maintainers.

See also:

- [Protected branches](#)
- [Pipeline security on protected branches](#)

Configure protected branches

You must have at least the *Maintainer* role.

1. Select *Menu* → *Projects* in the top bar and find your project.
2. In the left sidebar, select *Settings* → *Repository*.
3. Expand *Protected branches*.
4. In the *Protect branch...* drop-down list, select the branch you want to protect. Alternatively, you can use wildcards:

Wildcard	Examples
*-stage	#17-some-feature-stage, #42-other-feature-stage
production/*	production/app-server, production/load-balancer
app-server	app-server, production/app-server

5. Select a role from the *Allowed to merge*: drop-down list that is allowed to merge into this branch.
6. Select a role from the *Allowed to push*: drop-down list that is allowed to push into this branch.
7. Select *Protect*.
8. The protected branch is now displayed in the list of protected branches.

Merge requests

Merge requests allow you to check source code changes into a branch. When you open a merge request, you can visualise the code changes before merging and work on them together. Merge requests contain:

- A description of the request
- Code changes and code reviews
- Information about *CI/CD pipelines*
- discussion posts
- the list of commits

See also:

- [Merge requests](#)

Merge request workflows

1. You check out a new branch and submit your changes through a merge request.
2. You gather feedback from your team.
3. You work on the implementation and optimise the code with [code quality reports](#).
4. You verify your changes with [reports from unit tests](#) in *GitLab CI/CD*.
5. You avoid using dependencies whose licence is incompatible with your project with [licence compliance reports](#).
6. You request [approval](#) of your changes.
7. When the merge request is approved, *GitLab CI/CD* will deploy the changes to the production environment.

GitLab CI/CD

GitLab CI/CD can automatically build, test, deploy and monitor your applications during iterative code changes. This reduces the risk that you will develop new code based on buggy previous versions. In the process, little or no human intervention should be required from the development to its deployment of code changes.

The three main approaches to this continuous development are:

Continuous Integration

runs a series of scripts sequentially or in parallel that your application automatically builds and tests, for example after each `git pull` in a *feature branch*. This is to make it less likely that you will introduce bugs into your application.

If the checks work as expected, you can make a *merge request*; if the checks fail, you can revert the changes if necessary.

See also:

- [Continuous integration](#)

Continuous Delivery

goes one step further than Continuous Integration and also continuously deploys the application. However, this still requires manual intervention to manually deploy the changes to a *deployment branch*.

See also:

- [Continuous Delivery](#)
- [Continuous Delivery](#)

Continuous Deployment

also performs the deployment of the software to the productive infrastructure automatically.

Activating CI/CD in a project

1. Select *Menu* → *Projects* in the top bar and find your project.
2. In the left sidebar, select *Settings* → *General*.
3. Expand *Visibility, project features, permissions*.
4. In the *Repository* section, activate the *CI/CD option*.
5. Select *Save changes*.

CI/CD pipelines

Pipelines are the most important component of Continuous Integration, Delivery and Deployment.

Pipelines consist of:

Jobs

define what needs to be done, for example compiling code or testing.

See also:

[Jobs](#)

Stages

define when the jobs are to be executed, for example the phase `test` to be executed after the phase `build`.

See also:

Stages

Jobs are executed by so-called [runners](#). Several *jobs* in a *stage* are executed in parallel, provided there are enough simultaneous runners available.

If all *jobs* in a *stage* are successful, the pipeline continues with the next *stage*.

If a *job* in a *stage* fails, the next *stage* is normally not executed and the pipeline is terminated prematurely.

In general, pipelines are executed automatically and do not require any intervention once they have been created. However, there are cases where you can manually intervene in a pipeline.

A typical pipeline may consist of four *stages* that are executed in the following order:

1. A build stage with a job called `compile`.
2. A test stage with two parallel jobs called `unit-test` and `lint`.
3. A staging stage with a *job* called `deploy-to-stage`.
4. A production stage with a *job* called `deploy-to-prod`.

The corresponding `.gitlab-ci.yml` file could then look like this:

```
image: "docker.io/ubuntu"

stages:
  - build
  - test
  - staging
  - production

compile:
  stage: build
  script:
    - echo "Compiling the code..."
    - echo "Compile complete."

unit-test:
  stage: test
  script:
    - echo "Running unit tests... This will take about 60 seconds."
    - sleep 60
    - echo "Code coverage is 0%"

lint:
  stage: test
  script:
    - echo "Linting code... This will take about 10 seconds."
    - sleep 10
    - echo "No lint issues found."

deploy-to-stage:
  stage: stage
  script:
    - echo "Deploying application in staging environment..."
    - echo "Application successfully deployed to staging."
```

(continues on next page)

(continued from previous page)

```

deploy-to-production:
  stage: production
  script:
    - echo "Deploying application in production environment..."
    - echo "Application successfully deployed to production."

```

Show pipelines

You can find the current and historical pipeline runs on the *CI/CD* → *Pipelines* page of your project. You can also access pipelines for a *merge request* by navigating to their *Pipelines* tab. Select a pipeline to open the *Pipeline Details* page and view the jobs that have been run for that pipeline. From here you can cancel a running pipeline, retry *jobs* in a failed pipeline or delete a pipeline.

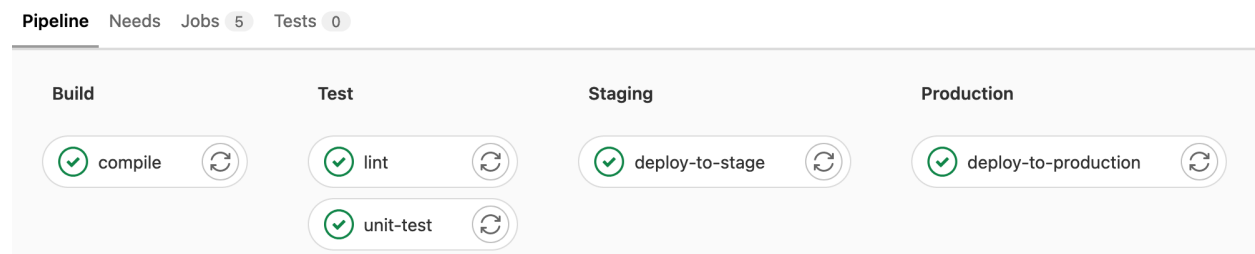


Fig. 3: GitLab CI/CD pipeline

See also:

- [Customize pipeline configuration](#)
- [Scheduled pipelines](#)
- [GitLab CI/CD variables](#)
- [Predefined variables reference](#)

Migrating GitHub Actions

GitLab CI/CD and GitHub Actions have some similarities in configuration, making migration to GitLab CI/CD relatively easy:

- Workflow configuration files are written in *YAML* and are stored in the repository along with the code.
- Workflows contain one or more jobs.
- Jobs include one or more steps or individual commands.
- Jobs can run on either managed or self-hosted machines.

However, there are also some differences, and this guide will show you the main differences so that you can migrate your workflow to GitLab CI/CD.

Jobs

Jobs in GitHub Actions are very similar to jobs in GitLab CI/CD. Both have the following characteristics:

- Jobs contain a series of steps or scripts that are executed in sequence.
- Jobs can be run on separate machines or in separate containers.
- Jobs are executed in parallel by default, but can also be configured to run sequentially.
- Jobs can execute a script or shell command, and in GitHub Actions all scripts are specified with the `run` key. In GitLab CI/CD, however, the script steps are specified with the `script` key.

Below is an example of the syntax of the two systems.

GitHub Actions syntax for jobs

```
jobs:
  my_job:
    steps:
      - uses: actions/checkout@v3
      - run: echo "Run my script here"
```

GitLab CI/CD syntax for jobs

```
my_job:
  variables:
    GIT_CHECKOUT: "true"
  script:
    - echo "Run my script here"
```

Runners

Runners are machines on which jobs are run. Both GitHub Actions and GitLab CI/CD offer managed and self-hosted variants of runners. In GitHub Actions, the `runs-on` key is used to run jobs on different platforms, while in GitLab CI/CD this is done with tags.

GitHub Actions syntax for Runner

```
my_job:
  runs-on: ubuntu-latest
  steps:
    - run: echo "Hello Pythonistas!"
```

GitLab CI/CD syntax for Runner

```
my_job:
  tags:
    - linux
  script:
    - echo "Hello Pythonistas!"
```

Docker images

GitHub Actions syntax for Docker images

```
jobs:
  my_job:
    container: python:3.10
```

GitLab CI/CD syntax for Docker images

```
my_job:
  image: python:3.10
```

See also:

- [Run your CI/CD jobs in Docker containers](#)

Syntax for conditions and expressions

GitHub Actions uses the `if` keyword to prevent a job from running if a condition is not met. GitLab CI/CD uses `rules` to determine whether a job is executed under a certain condition.

Below is an example of the syntax of the two systems.

GitHub syntax for conditions and expressions

```
jobs:
  deploy:
    if: contains( github.ref, 'main')
    runs-on: ubuntu-latest
    steps:
      - run: echo "Deploy to production server"
```

GitLab syntax for conditions and expressions

```
deploy:
  stage: deploy
  script:
    - echo "Deploy to production server"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
```

Besides if, GitLab also offers other rules such as `changes`, `exists`, `allow_failure`, `variables` and `when`.

See also:

- [rules](#)
- [Complex rules](#)

Dependencies between jobs

Both GitHub Actions and GitLab CI/CD allow you to set dependencies for a job. In both systems, jobs run in parallel by default, but GitLab CI/CD has a `stages` concept where jobs in one stage run concurrently, but the next stage does not start until all jobs in the previous stage have completed. In GitHub Actions, dependencies between jobs can be explicitly mapped with the `needs` key.

Below is an example of the syntax for each system. The workflows start with two jobs running in parallel named `unit-test` and `lint`. When these jobs are completed, another job called `deploy-to-stage` is run. Finally, when `deploy-to-stage` is complete, the job `deploy-to-prod` is executed.

GitHub Actions syntax for dependencies between jobs

```
jobs:
  unit-test:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Running unit tests... This will take about 60 seconds."
      - run: sleep 60
      - run: echo "Code coverage is 0%"

  lint:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Linting code... This will take about 10 seconds."
      - run: sleep 10
      - run: echo "No lint issues found."

  deploy-to-stage:
    runs-on: ubuntu-latest
    needs: [unit-test, lint]
    steps:
      - run: echo "Deploying application in staging environment..."
      - run: echo "Application successfully deployed to staging."
```

(continues on next page)

(continued from previous page)

```
deploy-to-prod:
  runs-on: ubuntu-latest
  needs: [deploy-to-stage]
  steps:
    - run: echo "Deploying application in production environment..."
    - run: echo "Application successfully deployed to production."
```

GitLab CI/CD syntax for dependencies between jobs

```
stages:
  - test
  - stage
  - prod

unit-test:
  stage: test
  script:
    - echo "Running unit tests... This will take about 60 seconds."
    - sleep 60
    - echo "Code coverage is 0%"

lint:
  stage: test
  script:
    - echo "Linting code... This will take about 10 seconds."
    - sleep 10
    - echo "No lint issues found."

deploy-to-stage:
  stage: stage
  script:
    - echo "Deploying application in staging environment..."
    - echo "Application successfully deployed to staging."

deploy-to-prod:
  stage: prod
  script:
    - echo "Deploying application in production environment..."
    - echo "Application successfully deployed to production."
```

Artefacts

Both GitHub Actions and GitLab CI/CD can upload files and directories created by a job as artefacts. These artefacts can be used to preserve data across multiple jobs.

Below is an example of the syntax for both systems.

GitHub Actions syntax for artefacts

```
- name: Archive code coverage results
uses: actions/upload-artifact@v3
with:
  name: code-coverage-report
  path: output/test/code-coverage.html
```

GitLab CI/CD syntax for artefacts

```
script:
artifacts:
  paths:
    - output/test/code-coverage.html
```

Databases and service containers

Both systems allow you to include additional containers for databases, caching or other dependencies.

GitHub Actions uses the `container` key, while in GitLab CI/CD a container for the job is specified with the `image` key. In both systems, additional service containers are specified with the `services` key.

Below is an example of the syntax of the two systems.

GitHub Actions syntax for databases and service containers

```
jobs:
  test:
    runs-on: ubuntu-latest

    services:
      postgres:
        image: postgres
        env:
          POSTGRES_USER: postgres
          POSTGRES_PASSWORD: postgres
          POSTGRES_DB: postgres
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5

    steps:
      - name: Python
        uses: actions/checkout@v3
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'
```

(continues on next page)

(continued from previous page)

```

- name: Test with pytest
  run: python -m pytest
  env:
    DATABASE_URL: 'postgres://postgres:postgres@localhost:${{ job.services.
↪postgres.ports[5432] }}/postgres'

```

GitLab CI/CD syntax for database and service containers

```

test:
  variables:
    POSTGRES_PASSWORD: postgres
    POSTGRES_HOST: postgres
    POSTGRES_PORT: 5432
  image: python:latest
  services:
    - postgres
  script:
    - python -m pytest

```

Mapping

GitHub	GitLab
Concepts	
actions/upload-artifact@v2	artifacts
actions/cache@v2	cache
actions/download-artifact@v2	dependencies
environment	environment
container	image
actions/deploy-pages@main	pages
actions/create-release@v1	release
run	script, after_scri
hashicorp/vault-action@v2.5.0	secrets
services	services
runs-on	tags
timeout-minutes	timeout
Environment variables	
\${{ github.api_url }}	CI_API_V4_URL
\${{ github.workspace }}	CI_BUILDS_DIR
\${{ github.ref }}	CI_COMMIT_BRANCH,
\${{ github.sha }}	CI_COMMIT_SHA, CI_
\${{ github.job }}	CI_JOB_ID, CI_JOB_
\${{ github.event_name == 'workflow_dispatch' }}	CI_JOB_MANUAL
\${{ job.status }}	CI_JOB_STATUS
\${{ github.server_url }}/\${{ github.repository }}	CI_MERGE_REQUEST_
\${{ github.token }}	CI_NODE_INDEX
\${{ strategy.job-total }}	CI_NODE_TOTAL

Table 1 – continued from p

GitHub	GitLab
<code>{{ github.repository }}/{{ github.workflow }}</code>	<code>CI_PIPELINE_ID</code>
<code>{{ github.workflow }}</code>	<code>CI_PIPELINE_IID</code>
<code>{{ github.event_name }}</code>	<code>CI_PIPELINE_SOURCE</code>
<code>{{ github.actions }}</code>	<code>CI_PIPELINE_TRIGGERED</code>
<code>{{ github.server_url }}/{{ github.repository }}/actions/runs/{{ github.run_id }}</code>	<code>CI_PIPELINE_URL</code>
<code>{{ github.workspace }}</code>	<code>CI_PROJECT_DIR</code>
<code>{{ github.repository }}</code>	<code>CI_PROJECT_ID</code> <code>CI_PROJECT_NAME</code>
<code>{{ github.event.repository.name }}</code>	<code>CI_PROJECT_NAME</code>
<code>{{ github.repository_owner }}</code>	<code>CI_PROJECT_NAMESP</code>
<code>{{ github.event.repository.full_name }}</code>	<code>CI_PROJECT_TITLE</code>
<code>{{ github.server_url }}/{{ github.repository }}</code>	<code>CI_PROJECT_URL</code>
<code>{{ github.event.repository.clone_url }}</code>	<code>CI_REPOSITORY_URL</code>
<code>{{ runner.os }}</code>	<code>CI_RUNNER_EXECUTABLE</code>
<code>{{ github.server_url }}</code>	<code>CI_SERVER_HOST</code> , <code>CI_SERVER_PROTOCOL</code>
<code>{{ github.actions }}</code>	<code>CI_SERVER</code> , <code>GITLAB_CI</code>
<code>{{ github.actor }}</code>	<code>GITLAB_USER_EMAIL</code>
<code>{{ github.event_path }}</code>	<code>TRIGGER_PAYLOAD</code>
<code>{{ github.event.pull_request.assignees }}</code>	<code>CI_MERGE_REQUEST_AUTHOR</code>
<code>{{ github.event.pull_request.number }}</code>	<code>CI_MERGE_REQUEST_IID</code>
<code>{{ github.event.pull_request.labels }}</code>	<code>CI_MERGE_REQUEST_LABELS</code>
<code>{{ github.event.pull_request.milestone }}</code>	<code>CI_MERGE_REQUEST_MILESTONE</code>
<code>{{ github.event.pull_request.head.ref }}</code>	<code>CI_MERGE_REQUEST_HEAD_REF</code>
<code>{{ github.event.pull_request.head.sha }}</code>	<code>CI_MERGE_REQUEST_HEAD_SHA</code>
<code>{{ github.event.pull_request.head.repo.full_name }}</code>	<code>CI_MERGE_REQUEST_HEAD_REPO_FULL_NAME</code>
<code>{{ github.event.pull_request.head.repo.url }}</code>	<code>CI_MERGE_REQUEST_HEAD_REPO_URL</code>
<code>{{ github.event.pull_request.base.ref }}</code>	<code>CI_MERGE_REQUEST_BASE_REF</code>
<code>{{ github.event.pull_request.base.sha }}</code>	<code>CI_MERGE_REQUEST_BASE_SHA</code>
<code>{{ github.event.pull_request.title }}</code>	<code>CI_MERGE_REQUEST_TITLE</code>
<code>{{ github.event.pull_request.number }}</code>	<code>CI_EXTERNAL_PULL_REQUEST_IID</code>
<code>{{ github.event.pull_request.head.repo.full_name }}</code>	<code>CI_EXTERNAL_PULL_REQUEST_REPO_FULL_NAME</code>
<code>{{ github.event.pull_request.base.repo.full_name }}</code>	<code>RCI_EXTERNAL_PULL_REQUEST_REPO_FULL_NAME</code>

GitLab Package Registry

You can also publish your distribution packages in the package registry of your GitLab project and use them with both [Pip](#) and [twine](#).

See also:

[GitLab Package Registry](#)

git-big-picture

git-big-picture visualises Git repositories as DAGs. The tool comes with some filters to show only the interesting areas, for example the hierarchy of tags and branches.

Examples

```
git big-picture -o git-big-picture.svg
```

```
$ git big-picture -ao git-big-picture_all.svg
```

Installation

You can easily install git-big-picture with:

```
$ pipenv install git-big-picture
Installing git-big-picture...
Adding git-big-picture to Pipfile's [packages]...
✓ Installation Succeeded
...
```

Git-Integration

You can easily integrate the tool into Git by adding the script git-big-picture to \$PATH. Then you can use it, for example with:

```
$ git big-picture -h
Usage: git-big-picture OPTIONS [<repo-directory>]

Options:
  --version          show program's version number and exit
  -h, --help         show this help message and exit
  --pstats=FILE      run cProfile profiler writing pstats output to FILE
  -d, --debug        activate debug output

Output Options:
  Options to control output and format

  -f FMT, --format=FMT
                        set output format [svg, png, ps, pdf, ...]
  -g, --graphviz      output lines suitable as input for dot/graphviz
  -G, --no-graphviz   disable dot/graphviz output
  -p, --processed     output the dot processed, binary data
  -P, --no-processed  disable binary output
  -v CMD, --viewer=CMD
                        write image to tempfile and start specified viewer
  -V, --no-viewer     disable starting viewer
  -o FILE, --outfile=FILE
```

(continues on next page)

(continued from previous page)

```

        write image to specified file
-O, --no-outfile    disable writing image to file

Filter Options:
Options to control commit/ref selection

-a, --all           include all commits
-b, --branches      show commits pointed to by branches
-B, --no-branches   do not show commits pointed to by branches
-t, --tags          show commits pointed to by tags
-T, --no-tags       do not show commits pointed to by tags
-r, --roots         show root commits
-R, --no-roots      do not show root commits
-m, --merges        include merge commits
-M, --no-merges     do not include merge commits
-i, --bifurcations  include bifurcation commits
-I, --no-bifurcations
                   do not include bifurcation commits

```

Configuration

The standard `git config` infrastructure can be used to configure `git-big-picture`. Most of the command line arguments can be configured in a `[big-picture]` section. For example, to configure `firefox` as a viewer with

```
$ git config --global big-picture.viewer firefox
```

will create the following section in your `~/.gitconfig` file:

```
[big-picture]
viewer = firefox
```

Note: However, this disables other options at the same time. For example, you can no longer display the graph with `Graphviz`:

```
$ git-big-picture -g
fatal: Options '-g | --graphviz' and '-p | --processed' are incompatible with other
↳ output options.
```

In this case you must also specify the `-V` or `--no-viewer` option:

```
$ git-big-picture -g -V
digraph {
    "c509669a01b156900eed9f1c9f927b6d2f7bb95b"[label="origin/pyup-scheduled-update-2020-
↳ 11-16", color="/pastel13/2", style=filled];
    ...
}
```

etckeeper

`etckeeper` is a collection of tools that can be used to manage the `/etc` directory in a Git repository. This allows changes to be checked and undone if necessary. It also connects to package managers such as `apt` to automatically commit changes made to `/etc` during a package upgrade. Finally, it also takes into account metadata of files that Git does not normally manage, but which are important for `/etc`, such as the permissions of `/etc/shadow`.

Installation

`etckeeper` can be easily installed with

```
$ sudo apt install git etckeeper
```

Configuration

1. The configuration of `etckeeper` is done in the `etckeeper.conf` file:

```
# The VCS to use.
#VCS="hg"
VCS="git"
#VCS="bazaar"
#VCS="darcs"
...
```

2. In addition, the following two automatic commits should be avoided:

```
# Uncomment to avoid etckeeper committing existing changes
# to /etc automatically once per day.
AVOID_DAILY_AUTOCOMMITS=1
...
# Uncomment to avoid etckeeper committing existing changes to
# /etc before installation. It will cancel the installation,
# so you can commit the changes by hand.
AVOID_COMMIT_BEFORE_INSTALL=1
```

3. Now git itself should be configured, see *Configuration*.
4. Finally, the `/etc` directory can be taken under Git version control with:

```
$ cd /etc/
$ sudo etckeeper init
Initialized empty Git repository in /etc/.git/
$ sudo etckeeper commit "Initial commit"
```

Use

If a configuration file is now edited, the changes can now be easily logged with Git.

Managing metadata

Since Git itself does not record complete metadata, etckeeper has set up a *pre-commit hook* in `/etc/.git/hooks/pre-commit`. This hook logs the `chmod` and `chgrp` entries for all files that do not correspond to the standard permissions in the file `/etc/.etckeeper`:

```
maybe chmod 0755 '.'
maybe chmod 0700 './.etckeeper'
maybe chmod 0644 './.gitignore'
...
. gitignore
```

Files that are not to be versioned with Git in the `/etc` directory can be added in the file `/etc/.gitignore`. This file is created when etckeeper is initiated and can be extended if necessary after the comment

```
# end section managed by etckeeper
```

Git's database internals

See also:

- [Commits are snapshots, not diffs](#)
- [Git's database internals](#)
 - [Part I: packed object store](#)
 - [Part II: commit history queries](#)
 - [Part III: file history queries](#)
 - [Part IV: distributed synchronization](#)
 - [Part V: scalability](#)

Git glossary

Branch

A branch is a development line. The last commit on a branch is called the tip of the branch, which is referenced by a head and which moves on as more development is done on the branch. A single Git repository can have any number of branches, but its *Working Tree* is associated with only one of them – the current or checked-out branch – and *HEAD* points to that branch.

Cache

Obsolete for [Index](#).

Clone

Local version of a repository including all commits and branches.

Commit

A snapshot of the entire Git repository, compressed in a [SHA](#).

Fork

A copy of a repository on [GitLab](#) that belongs to another user or group.

Git

Git is a distributed version control system.

GitLab

Web application for version management based on [git](#). Later, [GitLab CI/CD](#), a system for continuous integration, GitLab Runner, Container Registry and many other things were added.

See also:

- [GitLab](#)

HEAD

The HEAD pointer represents your current working directory and can be moved to different branches, tags or commits using `git switch`.

Index

A collection of files with status information whose content is saved as objects. The index is a saved version of your [Working Tree](#).

origin

The usual upstream repository. Most projects have at least one upstream project that they track. By default, `origin` is used for this purpose. New upstream updates are fetched into branches named `origin/NAME_OF_UPSTREAM_BRANCH`, which you can see with `git branch -r`.

Merge request

Place to compare and discuss the changes introduced in a branch with ratings, comments, tests ETC..

See also:

- [Merge requests](#).

Remote repository

shared repository, for example on [GitLab](#), for exchanging changes in a team.

Trunk-Based Development**TBD**

Git workflow with short-lived topic branches that are quickly merged into a single main branch.

See also:

- [Trunk Based Development](#)

Working Tree

The tree of the files actually checked out. The working tree normally contains the content of the [HEAD](#) commit tree as well as all local changes that you have made but not yet transferred.

7.2 Manage data with DVC

For data analysis, and especially machine learning, it is extremely valuable to be able to reproduce different versions of analyses that have been carried out with different data sets and parameters. However, in order to obtain reproducible analyses, both the data and the model (including the algorithms, parameters, etc.) must be versioned. Versioning data for reproducible analysis is a bigger problem than versioning models because of the size of the data. Tools like [DVC](#) help manage data by allowing users to transfer it to a remote data store using a [Git](#) like workflow. This simplifies the retrieval of certain versions of data in order to reproduce an analysis.

DVC was developed to be able to use ML models and data sets together and to manage them in a comprehensible manner. It works with different version managements, but does not need them. In contrast to [DataLad/git-annex](#), for example, it is not limited to Git as version management, but can also be used together with Mercurial, see github.com/crobarcro/dvc/dvc/scm.py. It also uses its own system for storing files with support for SSH and HDFS, among others.

DataLad, on the other hand, focuses more on discovering and consuming datasets, which are then easily managed with Git. DVC, on the other hand, stores each step in the pipeline in a separate `.dvc` file that can then be managed by Git.

These `.dvc` files, however, allow practical tools for manipulating and visualizing DAGs, see, for example, [visualisation of DAGs](#).

External dependencies can also be specified with *dvc remote*.

See also:

- [Tutorial](#)
- [Documentation](#)
- [Git Repository](#)

7.2.1 Installation

Finally, external dependencies can also be specified with Pipenv.

Note: You have to explicitly state the extras. This can be `[ssh]`, `[s3]`, `[gs]`, `[azure]`, and `[oss]` or `[all]`. For `ssh` the command looks like this:

```
$ pipenv install dvc[ssh]
```

Alternatively, DVC can also be installed via other package managers:

```
$ sudo wget https://dvc.org/deb/dvc.list -O /etc/apt/sources.list.d/dvc.list
$ sudo apt update
$ sudo apt install dvc
```

```
$ brew install iterative/homebrew-dvc/dvc
```

Note: The following example was created with a current DVC version (1.0.0a9), which partly uses a different syntax than earlier versions. You can currently (8th June 2020) only install this with `pip`:

```
$ pipenv install dvc[all]==1.0.0a9
```

Create a project

DVC can be easily initialised with:

```
$ mkdir -p dvc-example/data
$ cd dvc-example
$ git init
$ dvc init
$ git add .dvc
$ git commit -m "Initialise DVC"
```

dvc init

creates a directory `.dvc/` with config, `.gitignore` and cache directory.

git commit

puts `.dvc/config` and `.dvc/.gitignore` under version control.

Configure

Before DVC is used, even a remote storage is established. This should be accessible to everyone who should access the data or the model. It's similar to using a Git server. Often, however, this is also an NFS mount, which can be integrated as follows, for example:

```
$ sudo mkdir -p /var/dvc-storage
$ dvc remote add -d local /var/dvc-storage
Setting 'local' as a default remote.
$ git commit .dvc/config -m "Configure local remote"
[master efaeb84] Configure local remote
1 file changed, 4 insertions(+)
```

-d, --default

Default value for the space removed

local

Name of the remote location

/var/dvc-storage

URL of the remote location

In addition, other protocols are supported, which are preceded by the path, including `ssh:`, `hdfs:` and `https:`.

Another remote data storage can simply be added, for example with:

```
$ dvc remote add webserver https://dvc.example.org/myproject
```

The associated configuration file `.dvc/config` looks like this:

```
['remote "local"']
url = /var/dvc-storage
[core]
remote = local
['remote "webserver"']
url = https://dvc.example.org/myproject
```

Add data and directories

With DVC you can save and version files, ML models, directories and intermediate results with Git without having to check the file content into Git:

```
$ dvc get https://github.com/iterative/dataset-registry get-started/data.xml \
    -o data/data.xml
$ dvc add data/data.xml
```

This will add the file `data/data.xml` in `data/.gitignore` and write the meta information in `data/data.xml.dvc`. Further information on the file format of the `*.dvc` can be found under [DVC-File Format](#).

In order to be able to manage different versions of your project data with Git, you only have to add the CVS file:

```
$ git add data/.gitignore data/fortune500.csv.dvc
$ git commit -m "Add raw data to project"
```

Store and retrieve data

The data can be copied from the working directory of your Git repository to the remote storage space with

```
$ dvc push
```

If you want to call up more current data, you can do so with

```
$ dvc pull
```

Import and update

You can also import data and models from another project with the command `dvc import`, for example:

```
$ dvc import https://github.com/iterative/dataset-registry get-started/data.xml
Importing 'get-started/data.xml (https://github.com/iterative/dataset-registry)' ->
↪ 'data.xml'
```

This loads the file from the [dataset-registry](#) into the current working directory, adds `.gitignore` and creates `data.xml.dvc`.

With `dvc update` we can update these data sources before we reproduce a pipeline that depends on these data sources, for example

```
$ dvc update data.xml.dvc
Stage 'data.xml.dvc' didn't change.
Saving information to 'data.xml.dvc'.
```

Pipelines

Connect code and data

Commands like `dvc add`, `dvc push` and `dvc pull` can be made independently of changes in the Git repository and therefore only provide the basis for managing large amounts of data and models. In order to actually achieve reproducible results, code and data must be linked together.

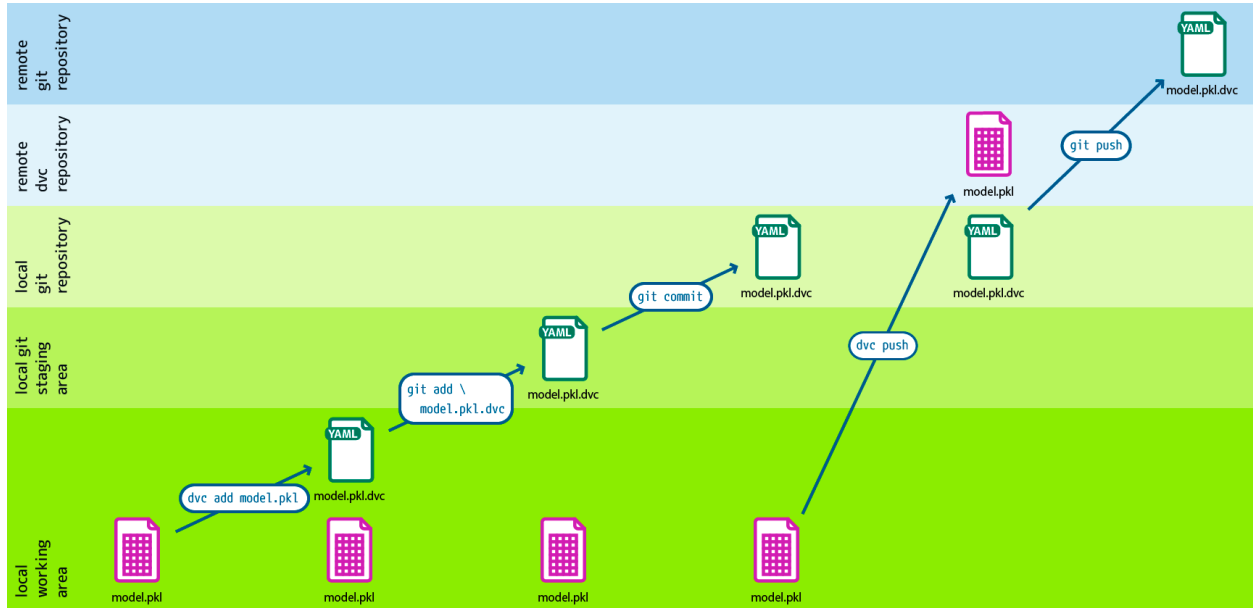


Fig. 4: Design: André Henze, Berlin

With `dvc run` you can create individual processing levels, each level being described by a source code file managed with Git as well as other dependencies and output data. All stages together then form the DVC pipeline.

In our example `dvc-example`, the first stage is to split the data into training and test data:

```
$ dvc run -n split -d src/split.py -d data/data.xml -o data/splitted \
python src/split.py data/data.xml
```

-n
indicates the name of the processing stage.

-d
dependencies on the reproducible command.

The next time `dvc repo` is called to reproduce the results, DVC checks these dependencies and decides whether they need to be updated or run again to get more current results.

-o
specifies the output file or directory.

In our case, the work area should have changed to:

```
.
├── data
│   ├── data.xml
│   └── data.xml.dvc
```

(continues on next page)

(continued from previous page)

```

+ |   └─ splitted
+ |       └─ test.tsv
+ |           └─ train.tsv
+ └─ dvc.lock
+   └─ dvc.yaml
+     └─ requirements.txt
+       └─ src
+         └─ split.py

```

The generated `dvc.yaml` file looks like this, for example:

```

stages:
  split:
    cmd: pipenv run python src/split.py data/data.xml
    deps:
      - data/data.xml
      - src/split.py
    outs:
      - data/splitted

```

Since the data in the output directory should never be versioned with Git, `dvc run` has already written the file `data/.gitignore`:

```

/data.xml
+ /splitted

```

Then the changed data only has to be transferred to Git or DVC:

```

$ git add data/.gitignore dvc.yaml
$ git commit -m "Create split stage"
$ dvc push

```

If several phases are now created with `dvc run` and the output of one command being specified as a dependency of another, a [DVC Pipeline](#) is created.

Parameterisation

In the next phase of our example, we parameterise the processing and create the file `params.yaml` with the following content:

```

max_features: 6000
ngram_range:
  lo: 1
  hi: 2

```

To read the parameters, the option `-p <filename>:<params_list>` must be added to the command `dvc run`, in our example:

```

$ dvc run -n featurise -d src/featurisation.py -d data/splitted \
  -p params.yaml:max_features,ngram_range.lo,ngram_range.hi -o data/features \
  python src/featurisation.py data/splitted data/features

```

This adds to the `dvc.yaml` file:

```

featurise:
  cmd: python src/featurization.py data/splitted data/features
  deps:
    - data/splitted
    - src/featurization.py
  params:
    - max_features
    - ngram_range.lo
    - ngram_range.hi
  outs:
    - data/features

```

So that this phase can be repeated, the MD5 hash values and parameter values are stored in the file `dvc.lock`:

```

featurise:
  cmd: python src/featurisation.py data/splitted data/features
  deps:
    - path: data/splitted
      md5: 1ce9051bf386e57c03fe779d476d93e7.dir
    - path: src/featurisation.py
      md5: a56570e715e39134adb4fdc779296373
  params:
    params.yaml:
      max_features: 1000
      ngram_range.hi: 2
      ngram_range.lo: 1

```

Finally `dvc.lock`, `dvc.yaml` and `data/.gitignore` in the Git repository need to be updated:

```
$ git add dvc.lock dvc.yaml data/.gitignore
```

See also:

- `dvc params`

Trial metrics

With the `dvc metrics` command, DVC is also a framework for recording and comparing the performance of experiments.

`evaluate.py` calculates the AUC (A rea U nder the C urve). It uses the test data set, reads the features from the file `features/test.pkl` and creates the metrics file `auc.metric`. It can be identified as a DVC metric with the `-M` option of `dvc run`, in our example with:

```
$ dvc run -n evaluate -d src/evaluate.py -d model.pkl -d data/features \
  -M auc.json python src/evaluate.py model.pkl data/features auc.json
```

```

evaluate:
  cmd: python src/evaluate.py model.pkl data/features auc.json
  deps:
    - data/features
    - model.pkl
    - src/evaluate.py
  metrics:

```

(continues on next page)

(continued from previous page)

```
- auc.json:
  cache: false
```

With `dvc metrics show` experiments can be compared then through various branches and tags:

```
$ dvc metrics show
  auc.json: 0.514172
```

Now to complete our first version of the DVC pipeline, let's add the files and a tag to the Git repository:

```
$ git add dvc.yaml dvc.lock auc.json
$ git commit -m 'Add stage <evaluate>'
$ git tag -a 0.1.0 -m "Initial pipeline version 0.1.0"
```

View pipelines

Such data pipelines can be displayed or represented as a dependency graph with `dvc dag`:

```
$ dvc dag

+-----+
| data/data.xml.dvc |
+-----+

      *
      *
      *

+-----+
| split |
+-----+

      *
      *
      *

+-----+
| featurize |
+-----+
**          **

**          *
*           **
+-----+      *
| train |      **
+-----+      *
**          **
**          **
*           *

+-----+
| evaluate |
+-----+

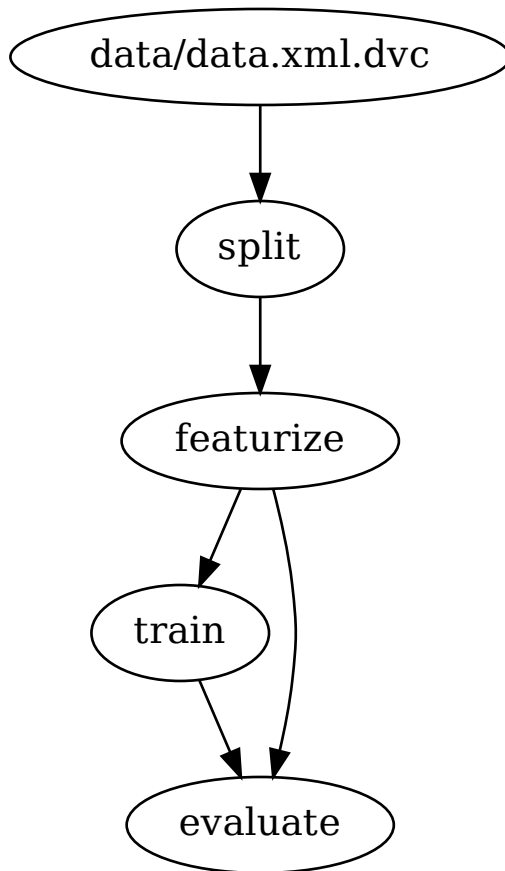
data/data.xml.dvc
prepare.dvc
featurize.dvc
```

(continues on next page)

(continued from previous page)

```
train.dvc  
evaluate.dvc
```

- With `dvc dag --dot` a `.dot` file for [Graphviz](#) is generated:



Reproduce

To reproduce the results of a project, we first clone the data managed with DVC:

```
$ git clone https://github.com/veit/dvc-example.git  
$ cd dvc-example  
$ dvc pull -TR  
A      data/data.xml  
1 file added  
$ ls data/  
data.xml  data.xml.dvc
```

Then you can easily reproduce the results with `dvc repro`:

```
$ dvc repro
Verifying data sources in stage: 'data/data.xml.dvc'
Stage 'split' didn't change, skipping
Stage 'featurize' didn't change, skipping
Stage 'train' didn't change, skipping
Stage 'evaluate' didn't change, skipping
```

You can now, for example, change parameters in the `params.yaml` file and then run through the pipeline again:

```
$ dvc repro
Stage 'data/data.xml.dvc' didn't change, skipping
Stage 'split' didn't change, skipping
Running stage 'featurize' with command:
    python src/featurization.py data/splitted data/features
...
Stage 'train' didn't change, skipping
Stage 'evaluate' didn't change, skipping
To track the changes with git, run:
    git add dvc.lock
```

In our case, changing the parameters had no effect on the result.

Note: DVC recognises changes to dependencies and outputs via md5 hash values in `dvc.lock`.

Vim and IDE integration

Vim

To recognize DVC files in Vim as YAML, you should add the following in `~/.vimrc`:

```
" DVC
autocmd! BufNewFile,BufRead Dvcfile,*.dvc setfiletype yaml
```

Visual Studio Code

For [Visual Studio Code](#), there is an extension for [DVC](#) that can be downloaded from the [Visual Studio Marketplace](#).

IntelliJ IDEs

[intellij-dvc](#) is a plugin for IntelliJ IDEs including PyCharm, IntelliJ IDEA and CLion. It can be downloaded from the [JetBrains Plugins-Repository](#).

FastDS

FastDS is an open source tool that combines *Git* and *DVC* to allow easy versioning of code and data.

Installation

FastDS can be easily installed with:

```
$ pipenv install fastds
```

Introduction

Even the creation of the initial repository is greatly simplified:

```
$ git init
$ dvc init
$ git add .
$ dvc add data/data.xml
$ git add data/.gitignore data/data.xml.dvc
$ git commit -m "Initial commit"
$ dvc push -r origin
$ git push origin
```

becomes:

```
$ fds init
$ fds add .
$ fds save -m "Initial commit"
```

FastDS abbreviates Git and DVC commands to minimise input errors and automate repetitive tasks:

init

initialises both the Git and DVC repositories.

status

returns the status of both repositories.

add

adds files to the Git or DVC repository.

commit

commits changes to the Git or DVC repository.

clone

clones the Git repository and fetches data from the remote DVC repository.

push

pushes data to the remote Git and DVC repositories.

save

adds changes to the project and commits them to the remote Git and DVC repositories.

7.3 Reproduce environments

Reproducible and secure Python environments are difficult to ensure. With the Python package manager [pip](#), the call would look like this:

```
$ python -m pip install --no-deps --require-hashes ----only-binary=:all:
```

Dedicated environments (for example with [Pipenv](#), [devpi](#) and [Spack](#) simplify this if you save the file with their specifications, for example `Pipfile`, `Pipfile.lock`, `package-lock.json` etc. In this way, you and others can reproduce the environments.

7.3.1 Spack

Modeling and simulation environments are very heterogeneous. [Spack](#) therefore supports many different production environments:

- 7 different compilers: Intel, GCC, Clang, PGI, ...
- Resolving dependencies
- Resolving different versions of dependencies

See also:

- [Docs](#)
- [Tutorial](#)
- [Spack Encyclopedia](#)
- [GitHub](#)

Previous systems

They usually do not offer any support for combinatorial versioning.

- Traditional binary package managers like RPM, yum, APT, yast, etc.
 - are designed to manage a single software stack
 - install one version of a package
 - usually problem-free upgrades to a stable, well-tested stack
- Port systems
 - BSD Ports, portage, NixOS, Macports, Homebrew, etc.
 - mostly little support for builds that are parameterised by compilers or dependent versions
- Virtual machines and Linux containers
 - Containers allow the creation of different environments for different applications
 - However, they do not solve the build problem for the image
 - Performance, security and upgrades become very complex with many different builds.

Spack installation

Requirements

- Interpreter for Spack:
 - Python 2.7 or Python 3.5–3.9
- Building software
 - C/C++ compilers
 - make, patch and bash
- Create and extract archives
 - tar, gzip and bzip
- Manage software repositories
 - git
- Sign and verify Build caches
 - gnupg2

```
$ sudo apt install build-essential patch tar gzip bzip2 git gnupg2
```

```
$ xcode-select --install
$ brew install make bash gzip bzip2 git gnupg
$ brew link gnupg
```

Installation

To install Spack the repository is cloned and then changed from the *develop* branch to the branch of the current release, in our case to *v0.17.1*:

```
$ git clone https://github.com/spack/spack.git
Cloning into 'spack'...
...
$ cd spack
$ git switch v0.19
```

Configure the shell

1. To configure the bash environment, the following is entered in the `~/.bashrc`:

```
export SPACK_ROOT=~/.spack
. $SPACK_ROOT/share/spack/setup-env.sh
```

2. The changed configuration is read with

```
$ source ~/.bashrc
```

Bootstrapping clingo

Spack uses **clingo** to resolve optimal versions and variants of dependencies when installing packages. To install clingo from pre-built binaries you can simply specify a package:

```
$ spack spec zlib
==> Bootstrapping clingo from pre-built binaries
==> Fetching https://mirror.spack.io/bootstrap/github-actions/v0.4/build_cache/linux-
centos7-x86_64-gcc-10.2.1-clingo-bootstrap-spack-idkenmhnsclju5gjghpcqa4h7o2a7aow.spec.
json
==> Fetching https://mirror.spack.io/bootstrap/github-actions/v0.4/build_cache/linux-
centos7-x86_64/gcc-10.2.1/clingo-bootstrap-spack/linux-centos7-x86_64-gcc-10.2.1-
clingo-bootstrap-spack-idkenmhnsclju5gjghpcqa4h7o2a7aow.spack
==> Installing "clingo-bootstrap@spack%gcc@10.2.1~docs~ipo+python+static_libstdcpp_
build_type=Release arch=linux-centos7-x86_64" from a buildcache
Input spec
-----
zlib

Concretized
-----
zlib@1.2.13%gcc@11.3.0+optimize+pic+shared build_system=makefile arch=linux-ubuntu22.04-
sandybridge
```

Note: When bootstrapping from pre-built binaries, Spack requires `patchelf` on Linux or `otool` on macOS. Otherwise Spack built it from sources and with a C++ compiler.

Bootstrap store

All tools Spack needs are installed in a separate store, which lives in the `$HOME/.spack` directory. The software installed there can be queried with:

```
$ spack find --bootstrap
==> Showing internal bootstrap store at "/srv/jupyter/.spack/bootstrap/store"
==> 3 installed packages
-- linux-rhel5-x86_64 / gcc@9.3.0 -----
clingo-bootstrap@spack  python@3.8

-- linux-ubuntu20.04-sandybridge / gcc@9.3.0 -----
patchelf@0.13
```

Compiler configuration

```
$ spack compilers
==> Available compilers
-- gcc ubuntu22.04-x86_64 -----
gcc@11.3.0
```

Build your own compiler

```
$ spack install gcc
...
==> gcc: Successfully installed gcc-11.2.0-azhiay4ugfrs634hqlez7u3f2li3wvzd
Fetch: 12.09s. Build: 2h 8m 13.92s. Total: 2h 8m 26.01s.
[+] /Users/veit/spack/opt/spack/darwin-bigsur-cannonlake/apple-clang-13.0.0/gcc-11.2.0-
    ↪ azhiay4ugfrs634hqlez7u3f2li3wvzd
```

However, Spack doesn't find the compiler at first:

```
$ spack compilers
==> Available compilers
-- gcc ubuntu20.04-x86_64 -----
gcc@9.3.0
```

Now, you can add the compiler with `spack compiler find`:

```
$ spack compiler find /srv/jupyter/spack/opt/spack/linux-ubuntu22.04-sandybridge/gcc-11.
    ↪ 3.0/gcc-12.2.0-gbaw464qxjuz6i3uud42cd5mb4xujxia/
==> Added 1 new compiler to /srv/jupyter/.spack/linux/compilers.yaml
gcc@12.2.0
==> Compilers are defined in the following files:
    /srv/jupyter/.spack/linux/compilers.yaml
```

`spack compilers` should now also find the newly installed compiler:

```
$ spack compilers
==> Available compilers
-- gcc ubuntu22.04-x86_64 -----
gcc@12.2.0 gcc@11.3.0
```

If you want to overwrite the default and site settings, you can edit `$HOME/.spack/packages.yaml`:

```
packages:
  all:
    compiler: [gcc@12.2.0]
```

PGP signing

Spack supports the signing and verification of packages with GPG keys. A separate key ring is used for Spack, why no keys are available from users' home directories.

When Spack is first installed, this key ring will be empty. The keys stored in `/var/spack/gpg` are the standard keys for a Spack installation. These keys are imported by `spack gpg init`. This will import the standard keys into the keyring as trusted keys.

Trust keys

Additional keys can be added to the key ring using `spack gpg trust <keyfile>`. Once a key is trusted, packages signed by the owner of that key can be installed.

Create a key

You can also create your own keys to be able to sign your own packages with

```
$ spack gpg export <location> [<key>...]
```

List keys

The keys available in the keyring can be listed with

```
$ spack gpg list
```

Remove a key

Keys can be removed with

```
$ spack gpg untrust <keyid>
```

Key IDs can be email addresses, names or fingerprints.

Combinatorial builds

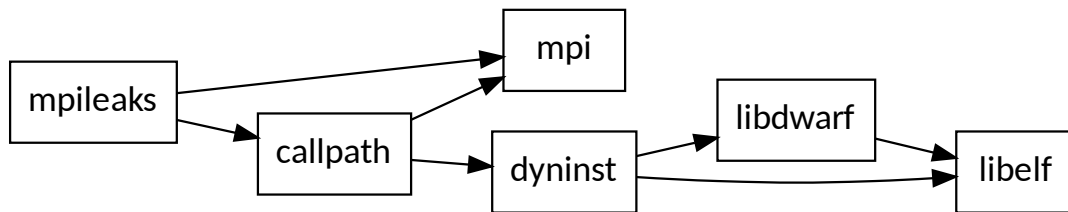
Environment modules

```
$ module avail
----- /opt/modules/modulefiles -----
acml-gnu/4.4 intel/12.0 mvapich2-pgi-ofa/1.7
acml-gnu_mp/4.4 intel/13.0 mvapich2-pgi-psm/1.7
acml-intel/4.4 intel/14.0(default) mvapich2-pgi-shmem/1.7...
$ module load intel/13.0
$ module load mvapich2-pgi-shmem/1.7
```

- Pros
 - replace different versions dynamically in the shell

- abstract a lot from the complexity of the environment
- Cons
 - Users need to keep in mind which versions of the build were made
 - It's easy to load the wrong module and cause a build to fail

Dependency DAG



Installation layout

```

$ tree /Users/veit/spack/opt/spack/
/Users/veit/spack/opt/spack/
├─ darwin-mojave-x86_64
│   ├── clang-10.0.1-apple
│   │   └─ autoconf-2.69-ymadj7a7gg52r76payi7jd7qu7qcuasp
│   │       └─ bin
│   │           ├── autoconf
│   │           └─ autoheader
│   ...

```

- Each unique dependency graph is given a unique configuration
- Each configuration is installed in a unique directory
 - Configurations of the same package coexist
- The hash value of a directed acyclic graph is appended
- Installed packages automatically find their dependencies
 - Spack embeds RPATH in binary files
 - There is no need to use modules or to set the LD_LIBRARY_PATH

spack list shows the available packages:

```

$ spack list
==> 3250 packages.
abinit                py-fiona
abyss                 py-fiscalyear

```

(continues on next page)

(continued from previous page)

```
accfft                                py-flake8
...
```

Spack provides a `spec` syntax for describing custom DAGs:

- without restrictions

```
$ spack install mpileaks
```

- @: custom version

```
$ spack install mpileaks@3.3
```

- %: custom compiler

```
$ spack install mpileaks@3.3 %gcc@4.7.3
```

- +/-: Build option

```
$ spack install mpileaks@3.3 %gcc@4.7.3 +threads
```

- =: Cross compile

```
$ spack install mpileaks@3.3 =bgq
```

- ^: Version of dependencies

```
$ spack install mpileaks %intel@12.1 ^libelf@0.8.12
```

- Spack ensures a configuration of each library per DAG
 - ensures the consistency of the Application Binary Interface (ABI)
 - The user does not need to know the DAG structure, just the names of the dependent libraries
- Spack can ensure that builds use the same compiler
- Different compilers can also be specified for different libraries of a DAG
- Spack can also provide ABI-incompatible, versioned interfaces such as the Message Passing Interface (MPI)
- For example, you can create `mpi` in different ways:

```
$ spack install mpileaks ^mvapich@1.9
$ spack install mpileaks ^openmpi@1.4
```

- Alternatively, Spack can also choose the right build himself if only the MPI 2 interface is implemented:

```
$ spack install mpileaks ^mpi@2
```

- Spack packages are simple Python scripts:

```
from spack import *

class Dyninst(Package):
    """API for dynamic binary instrumentation."""
    homepage = "https://paradyn.org"
```

(continues on next page)

(continued from previous page)

```

    version('8.2.1', 'abf60b7faabe7a2e', url="http://www.paradyn.org/release8.2/
↳DyninstAPI-8.2.1.tgz")
    version('8.1.2', 'bf03b33375afa66f', url="http://www.paradyn.org/release8.1.2/
↳DyninstAPI-8.1.2.tgz")
    version('8.1.1', 'd1a04e995b7aa709', url="http://www.paradyn.org/release8.1/
↳DyninstAPI-8.1.1.tgz")

    depends_on("libelf")
    depends_on("libdwarf")
    depends_on("boost@1.42:")

    def install(self, spec, prefix):
        libelf = spec['libelf'].prefix
        libdwarf = spec['libdwarf'].prefix

        with working_dir('spack-build', create=True):
            cmake('.',
                  '-DBoost_INCLUDE_DIR=%s' % spec['boost'].prefix.include,
                  '-DBoost_LIBRARY_DIR=%s' % spec['boost'].prefix.lib,
                  '-DBoost_NO_SYSTEM_PATHS=TRUE'
                  *std_cmake_args)
            make()
            make("install")

    @when('@:8.1')
    def install(self, spec, prefix):
        configure("--prefix=" + prefix)
        make()
        make("install")

```

- Dependencies in Spack can be optional:

- You can define *named variants*, for example in ~/spack/var/spack/repos/builtin/packages/vim/package.py:

```

class Vim(AutotoolsPackage):
    ...
    variant("python", default=False, description="build with Python")
    depends_on("python", when="+python")

    variant("ruby", default=False, description="build with Ruby")
    depends_on("ruby", when="+ruby")

```

- ... and use to install:

```

$ spack install vim +python
$ spack install vim -python

```

- Depending on other conditions, dependencies can optionally apply, for example gcc dependency on mpc from version 4.5:

```

depends_on("mpc", when="@4.5:")

```

- DAGs are not always complete before they are specified. Concretisations fill in the missing configuration details if you do not name them explicitly:

1. Normalisation

```
$ spack install mpileaks ^callpath@1.0+debug ^libelf@0.8.11
```

2. Specification

The detailed origin is saved with the installed package in `spec.yaml`:

```
spec:
- mpileaks:
  arch: linux-x86_64
  compiler:
    name: gcc
    version: 4.9.2
  dependencies:
    adept-utils: kszrtkpbzac3ss2ixcjkcorlaybnntp4
    callpath: bah5f4h4d2n47mgycej2mtrnrivvxy77
    mpich: aa4ar6ifj23yijqmdabeakpejcli72t3
  hash: 33hjjhxi7p6gyzn5ptgyes7sghyprujh
  variants: {}
  version: '1.0'
- adept-utils:
  arch: linux-x86_64
  compiler:
    name: gcc
    version: 4.9.2
  dependencies:
    boost: teesjv7ehpe5ksspjim5dk43a7qnowlq
    mpich: aa4ar6ifj23yijqmdabeakpejcli72t3
  hash: kszrtkpbzac3ss2ixcjkcorlaybnntp4
  variants: {}
  version: 1.0.1
- boost:
  arch: linux-x86_64
  compiler:
    name: gcc
    version: 4.9.2
  dependencies: {}
  hash: teesjv7ehpe5ksspjim5dk43a7qnowlq
  variants: {}
  version: 1.59.0
...
```

1. If unspecified, values based on the user settings are selected during the specification.
2. During the concretisation, new dependencies are added taking the constraints into account.
3. With the current algorithm, it is not possible to trace why a decision was made.
4. In the future there should be a full constraint solver.

Benefits of the build automation

- Spack makes it easy for teams to share their code
 - Recipes for common libraries
 - reduce the effort for reproducible builds
 - making it easier to share builds.
- Patches allow bug fixes to be provided quickly
 - Application developers who use a library often do not have write access to their repositories.
 - Library developers may not be able to fix problems as quickly as desired.
 - With Spack, application developers can quickly make corrections and undo changes.
- Python allows rapid adoption by development teams.
 - Many application developers are already familiar with Python.
 - The yaml syntax of the specs are expressive.

Use case 1: managing combinatorial installations

Display all installed configurations

```
$ spack find
==> 103 installed packages.
-- linux-x86_64 / gcc@4.8.2 -----
gdk-pixbuf@2.31.2  libpng@1.6.16      otf2@1.4          qhull@1.0
adept-utils@1.0.1  boost@1.55.0      cmake@5.6-special libdwarf@20130729  mpich@3.
  ↳0.4
adept-utils@1.0.1  cmake@5.6  dyninst@8.1.2  libelf@0.8.13  openmpi@1.8.2
-- linux-x86_64 / intel@14.0.2 -----
hwloc@1.9          mpich@3.0.4          starpu@1.1.4
-- linux-x86_64 / intel@15.0.0 -----
adept-utils@1.0.1  boost@1.55.0      libdwarf@20130729  libelf@0.8.13      mpich@3.
  ↳0.4
-- linux-x86_64 / intel@15.0.1 -----
adept-utils@1.0.1  callpath@1.0.2      libdwarf@20130729  mpich@3.0.4
boost@1.55.0      hwloc@1.9           libelf@0.8.13      starpu@1.1.4
```

- `spack find` shows all installed configurations
- There can also be different versions of the same package
- Packages are differentiated between architecture and compiler
- Spack also generates `modulefiles`, but these do not have to be used

Spack syntax to restrict the requests

```
$ spack find mpich
==> 5 installed packages.
-- linux-x86_64 / gcc@4.4.7 -----
mpich@3.0.4
-- linux-x86_64 / gcc@4.8.2 -----
mpich@3.0.4
-- linux-x86_64 / intel@14.0.2 -----
mpich@3.0.4
```

```
$ spack find libelf %intel
-- linux-x86_64 / intel@15.0.0 -----
libelf@0.8.13
-- linux-x86_64 / intel@15.0.1 -----
libelf@0.8.13
```

```
$ spack find libelf %intel@15.0.1
-- linux-x86_64 / intel@15.0.1 -----
libelf@0.8.13
```

Spack syntax for displaying the dependencies

```
$ spack find callpath
==> 2 installed packages.
-- linux-x86_64 / clang@3.4 -----      -- linux-x86_64 / gcc@4.9.2 -----
callpath@1.0.2                          callpath@1.0.2
```

```
$ spack find -dl callpath
==> 2 installed packages.
-- linux-x86_64 / clang@3.4 -----      -- linux-x86_64 / gcc@4.9.2 -----
xv2clz2      callpath@1.0.2              udltshts callpath@1.0.2
ckjazss      ^adept-utils@1.0.1          rfsu7fb ^adept-utils@1.0.1
3ws43m4      ^boost@1.59.0              ybet64y ^boost@1.55.0
ft7znm6      ^mpich@3.1.4                aa4ar6i ^mpich@3.1.4
qqnuet3      ^dyninst@8.2.1              tmnng5 ^dyninst@8.2.1
3ws43m4      ^boost@1.59.0              ybet64y ^boost@1.55.0
g65rdud      ^libdwarf@20130729          g2mxrl2 ^libdwarf@20130729
cj5p5fk      ^libelf@0.8.13              ynpai3j ^libelf@0.8.13
cj5p5fk      ^libelf@0.8.13              ynpai3j ^libelf@0.8.13
g65rdud      ^libdwarf@20130729          g2mxrl2 ^libdwarf@20130729
cj5p5fk      ^libelf@0.8.13              ynpai3j ^libelf@0.8.13
cj5p5fk      ^libelf@0.8.13              ynpai3j ^libelf@0.8.13
ft7znm6      ^mpich@3.1.4                aa4ar6i ^mpich@3.1.4
```

Use case 2: Python and other interpreted languages

```
$ spack install python@2.7.10
==> Building python.
==> Successfully installed python.
    Fetch: 5.01s. Build: 97.16s. Total: 103.17s.
[+] /srv/jupyterhub/spack/opt/spack/linux-x86_64/gcc-4.9.2/python-2.7.10-y2zr767
$ spack extensions python@2.7.10
==> python@2.7.10%gcc@4.9.2=linux-x86_64-y2zr767
==> 49 extensions:
geos                py-h5py            py-numpy           py-pypar           py-setuptools
libxml2             py-ipython         py-pandas          py-pyparsing       py-shiboken
py-basemap          py-libxml2         py-pexpect         py-pyqt            py-sip
py-biopython        py-lockfile        py-pil             py-pyside          py-six
py-cffi             py-mako            py-pmw             py-python-daemon   py-sphinx
py-cython           py-matplotlib      py-pychecker       py-pytz            py-sympy
py-dateutil         py-mock            py-pycparser       py-rpy2            py-virtualenv
py-epydoc           py-mpi4py          py-pyelftools      py-scientificpython py-yapf
py-genders          py-mx              py-pygments        py-scikit-learn    thrift
py-gnuplot          py-nose            py-pylint          py-scipy
==> 3 installed:
-- linux-x86_64 / gcc@4.9.2 -----
py-nose@1.3.6      py-numpy@1.9.2    py-setuptools@18.1
==> None currently activated.
```

```
$ spack activate py-numpy
==> Activated extension py-setuptools-18.1-gcc-4.9.2-ru7w3lx
==> Activated extension py-nose-1.3.6-gcc-4.9.2-vudjpw
==> Activated extension py-numpy-1.9.2-gcc@4.9.2-45hjzt
```

```
$ spack deactivate -a py-numpy
==> Deactivated extension py-numpy-1.9.2-gcc@4.9.2-45hjzt
==> Deactivated extension py-nose-1.3.6-gcc-4.9.2-vudjpw
==> Deactivated extension py-setuptools-18.1-gcc-4.9.2-ru7w3lx
```

Future features

- Lmod (Lua based module system) integration
- Resolve external dependencies
- Custom compiler flag injection
- XML test results (JUnit)

See also:

[Pull requests](#)

Use spack

List the available packages

```
$ spack list
==> 3247 packages.
abinit                py-fiona
abyss                 py-fiscalyear
...
```

or to filter for certain packages, for example

```
$ spack list numpy
==> 2 packages.
py-numpy  py-numpydoc
```

List the installed packages

```
$ spack find
==> 17 installed packages
-- darwin-mojave-x86_64 / clang@10.0.1-apple -----
bzip2@1.0.8  libffi@3.2.1  perl@5.26.2      python@3.7.4  zlib@1.2.11
diffutils@3.7 ncurses@6.1    pkgconf@1.6.1    readline@7.0
expat@2.2.5  openblas@0.3.6  py-numpy@1.16.4  sqlite@3.28.0
gdbm@1.18.1  openssl@1.1.1b  py-setuptools@41.0.1 xz@5.2.4
```

spack info

```
$ spack info py-numpy
PythonPackage:  py-numpy

Description:
  NumPy is the fundamental package for scientific computing with Python.
  It contains among other things: a powerful N-dimensional array object,
  sophisticated (broadcasting) functions, tools for integrating C/C++ and
  Fortran code, and useful linear algebra, Fourier transform, and random
  number capabilities

Homepage: http://www.numpy.org/

Tags:
  None

Preferred version:
  1.16.4  https://pypi.io/packages/source/n/numpy/numpy-1.16.4.zip

Safe versions:
  1.16.4  https://pypi.io/packages/source/n/numpy/numpy-1.16.4.zip
  1.16.3  https://pypi.io/packages/source/n/numpy/numpy-1.16.3.zip
```

(continues on next page)

(continued from previous page)

```

1.16.2 https://pypi.io/packages/source/n/numpy/numpy-1.16.2.zip
1.16.1 https://pypi.io/packages/source/n/numpy/numpy-1.16.1.zip
1.16.0 https://pypi.io/packages/source/n/numpy/numpy-1.16.0.zip
1.15.4 https://pypi.io/packages/source/n/numpy/numpy-1.15.4.zip
1.15.3 https://pypi.io/packages/source/n/numpy/numpy-1.15.3.zip
1.15.2 https://pypi.io/packages/source/n/numpy/numpy-1.15.2.zip
1.15.1 https://pypi.io/packages/source/n/numpy/numpy-1.15.1.zip
1.15.0 https://pypi.io/packages/source/n/numpy/numpy-1.15.0.zip
1.14.6 https://pypi.io/packages/source/n/numpy/numpy-1.14.6.zip
1.14.5 https://pypi.io/packages/source/n/numpy/numpy-1.14.5.zip
1.14.4 https://pypi.io/packages/source/n/numpy/numpy-1.14.4.zip
1.14.3 https://pypi.io/packages/source/n/numpy/numpy-1.14.3.zip
1.14.2 https://pypi.io/packages/source/n/numpy/numpy-1.14.2.zip
1.14.1 https://pypi.io/packages/source/n/numpy/numpy-1.14.1.zip
1.14.0 https://pypi.io/packages/source/n/numpy/numpy-1.14.0.zip
1.13.3 https://pypi.io/packages/source/n/numpy/numpy-1.13.3.zip
1.13.1 https://pypi.io/packages/source/n/numpy/numpy-1.13.1.zip
1.13.0 https://pypi.io/packages/source/n/numpy/numpy-1.13.0.zip
1.12.1 https://pypi.io/packages/source/n/numpy/numpy-1.12.1.zip
1.12.0 https://pypi.io/packages/source/n/numpy/numpy-1.12.0.zip
1.11.3 https://pypi.io/packages/source/n/numpy/numpy-1.11.3.zip
1.11.2 https://pypi.io/packages/source/n/numpy/numpy-1.11.2.zip
1.11.1 https://pypi.io/packages/source/n/numpy/numpy-1.11.1.zip
1.11.0 https://pypi.io/packages/source/n/numpy/numpy-1.11.0.zip
1.10.4 https://pypi.io/packages/source/n/numpy/numpy-1.10.4.zip
1.9.3 https://pypi.io/packages/source/n/numpy/numpy-1.9.3.zip
1.9.2 https://pypi.io/packages/source/n/numpy/numpy-1.9.2.zip
1.9.1 https://pypi.io/packages/source/n/numpy/numpy-1.9.1.zip

```

Variants:

Name [Default]	Allowed values	Description
blas [on]	True, False	Build with BLAS support
lapack [on]	True, False	Build with LAPACK support

Installation Phases:

```
build install
```

Build Dependencies:

```
blas lapack py-setuptools python
```

Link Dependencies:

```
blas lapack python
```

Run Dependencies:

```
python
```

Virtual Packages:

```
None
```

spack version

`spack version` shows the available versions, for example

```
$ spack versions python
==> Safe versions (already checksummed):
  3.7.4  3.7.0  3.6.5  3.6.1  3.5.1  3.3.6  2.7.15  2.7.11
  3.7.3  3.6.8  3.6.4  3.6.0  3.5.0  3.2.6  2.7.14  2.7.10
  3.7.2  3.6.7  3.6.3  3.5.7  3.4.10 3.1.5  2.7.13  2.7.9
  3.7.1  3.6.6  3.6.2  3.5.2  3.4.3  2.7.16  2.7.12  2.7.8
==> Remote versions (not yet checksummed):
  3.8.0b2  3.6.9    3.5.7rc1  3.5.0a2    3.4.0    3.1.2    2.7    2.4.3
  3.8.0b1  3.6.8rc1  3.5.6rc1  3.5.0a1    3.3.7rc1 3.1.1    2.6.9  2.4.2
...
```

Installation of certain packages

for example:

```
$ spack install python@3.7.4
```

or to install `py-numpy` for Python 3.7.4:

```
$ spack install py-numpy ^python@3.7.4
```

Then the installation can be checked with

```
$ spack find --deps py-numpy
==> 1 installed package
-- darwin-mojave-x86_64 / clang@10.0.1-apple -----
  py-numpy@1.16.4
    ^openblas@0.3.6
    ^python@3.7.4
      ^bzip2@1.0.8
      ^expat@2.2.5
      ^gdbm@1.18.1
        ^readline@7.0
          ^ncurses@6.1
      ^libffi@3.2.1
      ^openssl@1.1.1b
        ^zlib@1.2.11
      ^sqlite@3.28.0
      ^xz@5.2.4
```

Uninstall

```
$ spack uninstall py-numpy
```

or

```
$ spack uninstall --dependents py-numpy
```

Extensions and Python support

The Spack installation model assumes that each package lives in its own installation prefix. Modules in interpreted languages such as Python are typically installed in `$prefix/lib/python-3.7/site-packages/`, for example `/Users/veit/spack/opt/spack/darwin-mojave-x86_64/clang-10.0.1-apple/py-numpy-1.16.4-45sqnufha2yprpx6rxyelsokky65ucdy/lib/python3.7/site-packages/numpy`. However, packages installed in a different prefix can also be used. Such a package is called an *extension* in Spack.

Suppose Python was installed with

```
$ spack find python
==> 1 installed package
-- darwin-mojave-x86_64 / clang@10.0.1-apple -----
python@3.7.4
```

so *Extensions* can be found with

```
$ spack extensions python
==> python@3.7.4%clang@10.0.1-apple+bz2+ctypes+dbm+lzma~nis~optimizations_
↳ patches=210df3f28cde02a8135b58cc4168e70ab91dbf9097359d05938f1e2843875e57_
↳ +pic+pyexpat+pythoncmd+readline~shared+sqlite3+ssl~tix~tkinter~ucs4~uuid+zlib_
↳ arch=darwin-mojave-x86_64/jqlxzx
==> 623 extensions:
adios2                                py-munch
antlr                                 py-mx
...

==> 2 installed:
-- darwin-mojave-x86_64 / clang@10.0.1-apple -----
py-numpy@1.16.4  py-setuptools@41.0.1

==> None activated.
```

numpy can be added to the PYTHONPATH of the current shell with load:

```
$ spack load python
$ spack load py-numpy
$ python
Python 3.7.4 (default, Jul 28 2019, 20:00:06)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>>
```

Often, however, certain packages should be permanently available to a Python installation. Spack offers activate for this:

```
$ spack activate py-numpy
==> Activating extension py-numpy@1.16.4%clang@10.0.1-apple+blas+lapack arch=darwin-
    ↳mojave-x86_64/45sqnuf for python@3.7.4%clang@10.0.1-apple+bz2+ctypes+dbm+lzma~nis~
    ↳optimizations patches=210df3f28cde02a8135b58cc4168e70ab91dbf9097359d05938f1e2843875e57_
    ↳+pic+pyexpat+pythoncmd+readline~shared+sqlite3+ssl~tix~tkinter+ucs4~uuid+zlib_
    ↳arch=darwin-mojave-x86_64/jqlxzip
```

Environments, spack.yaml and spack.lock

1. Create a virtual environment:

```
$ spack env create python-311
==> Created environment 'python-311' in /srv/jupyter/spack/var/spack/environments/
    ↳python-311
==> You can activate this environment with:
==>   spack env activate python-311
```

Alternatively, it can also be saved in any other location, for example:

```
$ cd spackenvs/
$ spack env create -d python-311
==> Created environment in /srv/jupyter/jupyter-tutorial/spackenvs/python-311
==> You can activate this environment with:
==>   spack env activate /srv/jupyter/jupyter-tutorial/spackenvs/python-311
```

2. Check the virtual environment:

```
$ spack env list
==> 1 environments
    python-311
```

3. Activate the virtual environment:

```
$ spack env activate python-311
```

4. Check activation:

If you have activated an environment, you will only see what is in the current environment. That shouldn't be anything immediately after activation:

```
$ spack find
==> In environment python-311
==> No root specs
==> 0 installed packages
```

And if you want to check what environment you are in, you can query this with:

```
$ spack env status
==> In environment python-311
```

5. Finally, you can leave the activated environment with `spack env deactivate` or briefly `despacktivate`.

```
$ despacktivate
$ spack env status
==> No active environment
```

Install packages

```
$ spack env activate python-311
$ spack add python@3.11.0
$ spack install
==> Concretized python@3.11.0
- 4nvposf python@3.11.0%gcc@11.3.0+bz2+ctypes+dbm~debug+libxml2+lzma~nis~
  ↳optimizations+pic+pyexpat+pythoncmd+readline+shared+sqlite3+ssl~tix~tkinter~
  ↳ucs4+uuid+zlib build_system=generic patches=13fa8bf,b0615b2,f2fd060 arch=linux-
  ↳ubuntu22.04-sandybridge
- 6fefzf3 ^bz2@1.0.8%gcc@11.3.0~debug~pic+shared build_system=generic
  ↳arch=linux-ubuntu22.04-sandybridge
- 27f7g74 ^diffutils@3.8%gcc@11.3.0 build_system=autotools arch=linux-
  ↳ubuntu22.04-sandybridge
...
==> python: Successfully installed python-3.11.0-4nvposf6bicz5ogp6nqacfo4dfvwm7zv
Fetch: 5.19s. Build: 3m 48.84s. Total: 3m 54.03s.
[+] /srv/jupyter/spack/opt/spack/linux-ubuntu22.04-sandybridge/gcc-11.3.0/python-3.11.0-
  ↳4nvposf6bicz5ogp6nqacfo4dfvwm7zv
==> Updating view at /srv/jupyter/python-311/.spack-env/view
$ spack find
==> In environment /home/veit/python-311
==> Root specs
python@3.11.0
==> Installed packages
-- linux-ubuntu22.04-sandybridge / gcc@11.3.0 -----
berkeley-db@18.1.40      libiconv@1.16      readline@8.1.2
bzip2@1.0.8             libmd@1.0.4        sqlite@3.39.4
ca-certificates-mozilla@2022-10-11 libxml2@2.10.1     tar@1.34
diffutils@3.8           ncurses@6.3        util-linux-uuid@2.38.1
expat@2.4.8             openssl@1.1.1s     xz@5.2.7
gdbm@1.23               perl@5.36.0        zlib@1.2.13
gettext@0.21.1          pigz@2.7           zstd@1.5.2
libbsd@0.11.5           pkgconf@1.8.0
libffi@3.4.2            python@3.11.0
==> 25 installed packages
```

With `spack cd -e python-311` you can change to this directory, for example:

```
$ spack cd -e python-311
$ pwd
/srv/jupyter/spack/var/spack/environments/python-311
```

There you will find the two files `spack.yaml` and `spack.lock`.

`spack.yaml`

is the configuration file for the virtual environment. It is created in `~/spack/var/spack/environments/` when you call `spack env create`.

As an alternative to `spack install`, Python and other packages can also be installed by adding them to the specs list in `spack.yaml`:

```
specs: [python@3.11.0, ...]
```

concretization

The specifications can be made either `separately` or `together`. When concretising specs together the entire set of specs will be re-concretised after any addition of new user specs, to ensure the environment remains consistent.

view

True is the default value and equivalent to:

```
default:
  root: .spack-env/view
```

See also:

- `spack.yaml`

spack.lock

With `spack install` the specs are concretised, written in `spack.lock` and installed. In contrast to `spack.yaml` `spack.lock` is written in json format and contains the necessary information to be able to create reproducible builds of the environment:

```
{
  "_meta": {
    "file-type": "spack-lockfile",
    "lockfile-version": 4,
    "specfile-version": 3
  },
  "roots": [
    {
      "hash": "4nvposf6bicz5ogp6nqacfo4dfvwm7zv",
      "spec": "python@3.11.0"
    }
  ],
  "concrete_specs": {
    "4nvposf6bicz5ogp6nqacfo4dfvwm7zv": {
      "name": "python",
      "version": "3.11.0",
      "arch": {
        "platform": "linux",
        "platform_os": "ubuntu22.04",
        "target": {
          "name": "sandybridge",
          "vendor": "GenuineIntel",
          "features": [
            "aes",
            "avx",
            ...
          ]
        }
      }
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

    }
}

```

Installation of additional packages

Additional packages can be installed in the virtual environment with `spack add` and `spack install`. For [Matplotlib](#) it looks like this:

```

$ spack add py-numpy
==> Adding py-numpy to environment /srv/jupyter/jupyter-tutorial/spackenvs/python-311
$ spack install
==> Concretized python@3.11.0
[+] 4nvposf python@3.11.0%gcc@11.3.0+bz2+ctypes+dbm~debug+libxml2+lzma~nis~
↳optimizations+pic+pyexpat+pythoncmd+readline+shared+sqlite3+ssl~tix~tkinter~
↳ucs4+uuid+zlib build_system=generic patches=13fa8bf,b0615b2,f2fd060 arch=linux-
↳ubuntu22.04-sandybridge
[+] 6fefzf3 ^bzip2@1.0.8%gcc@11.3.0~debug~pic+shared build_system=generic
↳arch=linux-ubuntu22.04-sandybridge
[+] 27f7g74 ^diffutils@3.8%gcc@11.3.0 build_system=autotools arch=linux-
↳ubuntu22.04-sandybridge
...
==> Installing environment /srv/jupyter/jupyter-tutorial/spackenvs/python-311
...
==> Successfully installed py-numpy

```

Note: If a *Pipenv environment* has already been derived from this Spack environment, it must be rebuilt in order to receive the additional Spack package:

```

$ pipenv install --python=/srv/jupyter/spack/var/spack/environments/python-311/.spack-
↳env/view/bin/python
Creating a virtualenv for this project...
Pipfile: /srv/jupyter/jupyter-tutorial/pipenvs/python-311/Pipfile
Using /srv/jupyter/spack/var/spack/environments/python-311/.spack-env/view/bin/python
↳(3.11.0) to create virtualenv...
Creating virtual environment...Using base prefix '/srv/jupyter/jupyterhub/spackenvs/
↳python-374/.spack-env/view'
creator Venv(dest=/srv/jupyter/.local/share/virtualenvs/python-311-aGnPz55z,
↳clear=False, no_vcs_ignore=False, global=False, describe=CPython3Posix)
seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle,
↳via=copy, app_data_dir=/srv/jupyter/.local/share/virtualenv)
added seed packages: pip==22.3.1, setuptools==65.5.1, wheel==0.38.4
activators BashActivator,CShellActivator,FishActivator,NushellActivator,
↳PowerShellActivator,PythonActivator
✓ Successfully created virtual environment!
Virtualenv location: /srv/jupyter/.local/share/virtualenvs/python-311-aGnPz55z
Creating a Pipfile for this project...
Pipfile.lock not found, creating...
Locking [packages] dependencies...
Locking [dev-packages] dependencies...
Updated Pipfile.lock

```

(continues on next page)

(continued from previous page)

```
→(a3aa656db1de341c375390e74afd03f09eb681fe6881c58a71a85d6e08d77619)!  
Installing dependencies from Pipfile.lock (d77619)...  
To activate this project's virtualenv, run pipenv shell.  
Alternatively, run a command inside the virtualenv with pipenv run.
```

The installation can then be checked with:

```
$ pipenv run python  
Python 3.11.0 (main, Nov 19 2022, 11:29:15) [GCC 12.2.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import matplotlib.pyplot as plt
```

Configuration

`spack spec` specifies the dependencies of certain packages, for example

```
$ spack spec py-matplotlib  
Input spec  
-----  
py-matplotlib  
  
Concretized  
-----  
py-matplotlib@3.6.2%gcc@11.3.0~animation~fonts~latex~movies backend=agg build_  
→system=python_pip arch=linux-ubuntu22.04-sandybridge  
  ^freetype@2.11.1%gcc@11.3.0 build_system=autotools arch=linux-ubuntu22.04-  
→sandybridge  
    ^bzip2@1.0.8%gcc@11.3.0~debug~pic+shared build_system=generic arch=linux-  
→ubuntu22.04-sandybridge  
      ^diffutils@3.8%gcc@11.3.0 build_system=autotools arch=linux-ubuntu22.04-  
→sandybridge  
        ^libpng@1.6.37%gcc@11.3.0 build_system=autotools arch=linux-ubuntu22.04-sandybridge  
        ...
```

With `spack config get` you can look at the configuration of a certain environment:

```
$ spack config get  
# This is a Spack Environment file.  
#  
# It describes a set of packages to be installed, along with  
# configuration settings.  
spack:  
  # add package specs to the `specs` list  
  specs: [python@3.11.0, py-numpy]  
  view: true  
  concretizer:  
    unify: true
```

With `spack config edit` the configuration file `spack.yaml` can be edited.

Note: If packages are already installed in the environment, all dependencies should be specified again with `spack concretize -f`.

Loading the modules

With `spack env loads -r <env>` all modules are loaded with their dependencies.

Note: However, this does not currently work when loading modules from environments that are not in `$SPACK_ROOT/var/environments`.

Therefore we replace the directory `$SPACK_ROOT/var/environments` with a symbolic link:

```
$ rm $SPACK_ROOT/var/environments
$ cd $SPACK_ROOT/var/
$ ln -s /srv/jupyter/jupyter-tutorial/spackenvs environments
```

See also:

- [Environments Tutorial](#)

Spack mirrors

Some machines may not have internet access to get packages. Then you will need a local repository of tarballs from which to retrieve your files. Spack supports this with *Spack mirrors*. A mirror is a URL that points to a directory on the local file system or on a server and contains tarballs for all Spack packages.

Here is an example of the directory structure of a mirror:

```
$ tree /path/to/mirror/
/path/to/mirror/
├── autoconf
│   └── autoconf-2.69.tar.gz
├── automake
│   └── automake-1.16.1.tar.gz
├── bzip2
│   └── bzip2-1.0.8.tar.gz
├── diffutils
│   └── diffutils-3.7.tar.xz
├── expat
│   └── expat-2.2.5.tar.bz2
├── gcc
│   └── gcc-9.1.0.tar.xz
└── ...
```

`spack mirror create`

You can create a mirror with the command `spack mirror create`, provided you are on a machine that can access the Internet. The command iterates through all of Spack's packages and downloads the ones you want.

`spack mirror add`

Once you've created a mirror, you need to let Spack know about it. It's relatively easy. First find out the URL of your mirror. If it's a directory, you can use a file url like this:

```
$ spack mirror add local_filesystem file://$HOME/spack-mirror
```

Order of mirrors

`spack mirror ad` adds a line in `~/.spack/mirrors.yaml`:

```
mirrors:
  local_filesystem: file:///home/veit/spack-mirror
  remote_server: https://spack-mirror.cusy.io
```

If you want to change the order in which mirrors are searched for packages, you can edit this file and rearrange the sections: Spack searches them from top to bottom until a suitable entry is found.

Local default cache

Spack creates a cache for resources that are downloaded as part of installations. This cache is a valid Spack mirror: it uses the same directory structure and naming scheme as other Spack mirrors. The mirror is managed locally in the Spack installation directory at `~/spack/var/spack/cache/`.

7.3.2 Pipenv

Pipenv is a Python package manager. He uses **Pip** to install Python packages, but also simplifies the management and maintenance of dependencies.

Installation

This section covers the basics of installing **Python packages**.

Requirements for installing packages

Before installing Python packages, a few prerequisites must be met.

1. Make sure you are using the version of Python you want:

```
$ python --version
Python 3.10.6
```

Note: In **iPython** or a **Jupyter Notebook** you can find out the version with:

```
In [1]: import sys
        sys.version_info
sys.version_info(major=3, minor=10, micro=6, releaselevel='final', serial=0)
```

Note: If you use the system Python of your Linux distribution, you should first create a virtual environment with Python 3 and [Pip](#).

2. Make sure [Pip](#) is installed:

```
$ pip --version
pip 22.0.2 from /usr/lib/python3/dist-packages/pip (python 3.10)
```

1. If Pip is not yet installed, you can install it

```
$ sudo apt install python3-venv python3-pip
```

```
$ sudo apt install python-pip
```

Install Pipenv

[pipenv](#) is a dependency manager for Python projects. It to install Python packages, but it simplifies dependency management. Pip can be used to install Pipenv, but the `--user` flag should be used so that it is only available to that user. This is to prevent system-wide packages from being accidentally overwritten:

```
$ python3 -m pip install --user pipenv
...
Successfully installed distlib-0.3.4 filelock-3.4.2 pipenv-2022.1.8 platformdirs-2.4.1
↳ virtualenv-20.13.0 virtualenv-clone-0.5.7
```

Note: If pipenv is not available in the shell after the installation, the `USER_BASE/bin` directory may have to be specified in `PATH`.

The `USER_BASE` can be determined with:

```
$ python3 -m site --user-base
/srv/jupyter/.local
```

Then the `bin` directory must be appended and added to `PATH`. Alternatively, `PATH` can be set permanently by changing `~/.profile` or `~/.bash_profile`, in my case:

```
export PATH=/srv/jupyter/.local/bin:$PATH
```

The directory can be determined with `py -m site --user-site` and then `site-packages` can be replaced by `Scripts`. this then gives, for example:

```
C:\Users\veit\AppData\Roaming\Python38\Scripts
```

In order to be permanently available, this path can be entered in `PATH` in the control panel.

See also:

Further information on user-specific installations can be found in [User Installs](#).

Create virtual environments

Python virtual environments allow Python packages to be installed in an isolated location for a specific application, rather than installing them globally. So you have your own installation directories and do not share libraries with other virtual environments:

```
$ mkdir myproject
$ cd !$
cd myproject
$ pipenv install requests
Creating a virtualenv for this project...
...
Virtualenv location: /srv/jupyter/.local/share/virtualenvs/myproject-CZKj6mqJ
Creating a Pipfile for this project...
Installing requests...
Adding requests to Pipfile's [packages]...
...
```

Usage

Example

Now that `requests` is installed, it can be used.

1. As an example, we create the file `main.py` with the following content:

```
import requests

response = requests.get("https://cusy.io")

print(response.status_code)
```

1. Then the script can be executed with:

```
$ pipenv run python main.py
```

1. As a result of the call you should receive the HTTP status code `200`.

Using `pipenv run` ensures that your installed packages are available for your script.

Alternatively, you can also create a new shell `pipenv shell` with which all installed packages can be accessed:

```
$ pipenv shell
Launching subshell in virtual environment...
. /srv/jupyter/.local/share/virtualenvs/myproject-CZKj6mqJ/bin/activate
```

Options

-venv

specifies the path to the Virtualenv, usually in `~/ .local/share/virtualenvs/`. However, if you have created a directory `myproject/ .venv`, `pipenv` use this folder to create the associated Python environment there.

--py

specifies the path to the Python interpreter.

--envs

outputs options of the environment variables.

For `PIPENV_DONT_LOAD_ENV`, `PIPENV_DONT_USE_PYENV` and `PIPENV_DOTENV_LOCATION` see [Environment variables](#).

If you want to set these environment variables per project, you can use [direnv](#).

Also note that `pip` itself supports environment variables in case you need additional adjustments: [Pip Environment Variables](#).

Here is another example:

```
$ PIP_INSTALL_OPTION="-- -DCMAKE_BUILD_TYPE=Release" pipenv install -e .
```

Further information can be found at [Configuration With Environment Variables](#)

--three, --two, --python

uses Python 2 or Python 3 or a specific Python to which the path is given.

--site-packages

enables site packages for the virtual environment.

--pypi-mirror

indicates a PyPI mirror. The standard is the [Python Package Index \(PyPI\)](#).

However, you can also specify your own mirrors:

- with the environment variable `PIPENV_PYPI_MIRROR`
- in the command line, for example with:

```
$ pipenv install --pypi-mirror https://pypi.cusy.io/simple
$ pipenv update --pypi-mirror https://pypi.cusy.io/simple
...
```

- or in `pipfile`:

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true
name = "pypi"

[[source]]
url = "https://pypi.cusy.io/simple"
verify_ssl = true
name = "cusy-mirror"

[dev-packages]
```

(continues on next page)

(continued from previous page)

```
[packages]
requests = {version="*", index="cusey-mirror"}
maya = {version="*", index="pypi"}
records = {"*"}
```

Note: If a private index is used, there are currently still problems with hashing the packages.

You can find more options at [pipenv](#).

check

`pipenv check` checks for security holes and for **PEP 508** markers in the `pip` file. For this it uses [safety](#).

Example:

```
$ pipenv install django==1.10.1
Installing django==1.10.1...
...
$ pipenv check
Checking PEP 508 requirements...
Passed!
Checking installed package safety...

33075: django >=1.10,<1.10.3 resolved (1.10.1 installed)!
Django before 1.8.x before 1.8.16, 1.9.x before 1.9.11, and 1.10.x before 1.10.3, when
↳ settings.DEBUG is True, allow remote attackers to conduct DNS rebinding attacks by
↳ leveraging failure to validate the HTTP Host header against settings.ALLOWED_HOSTS.

33076: django >=1.10,<1.10.3 resolved (1.10.1 installed)!
Django 1.8.x before 1.8.16, 1.9.x before 1.9.11, and 1.10.x before 1.10.3 use a
↳ hardcoded password for a temporary database user created when running tests with an
↳ Oracle database, which makes it easier for remote attackers to obtain access to the
↳ database server by leveraging failure to manually specify a password in the database
↳ settings TEST dictionary.

33300: django >=1.10,<1.10.7 resolved (1.10.1 installed)!
CVE-2017-7233: Open redirect and possible XSS attack via user-supplied numeric redirect
↳ URLs

=====

Django relies on user input in some cases (e.g.
:func:`django.contrib.auth.views.login` and :doc:`i18n </topics/i18n/index>`)
to redirect the user to an "on success" URL. The security check for these
redirects (namely ``django.utils.http.is_safe_url()``) considered some numeric
URLs (e.g. ``http:999999999``) "safe" when they shouldn't be.

Also, if a developer relies on ``is_safe_url()`` to provide safe redirect
targets and puts such a URL into a link, they could suffer from an XSS attack.

CVE-2017-7234: Open redirect vulnerability in ``django.views.static.serve()``
```

(continues on next page)

(continued from previous page)

```
=====
```

A maliciously crafted URL to a Django site using the
`:func:`~django.views.static.serve`` view could redirect to any other domain. The
view no longer does any redirects as they don't provide any known, useful
functionality.

Note, however, that this view has always carried a warning that it is not
hardened for production use and should be used only as a development aid.

Note: `Pipenv` embeds an API client key from [pyup.io](https://pypi.org/project/pyupio/), instead of including a full copy of the [CC BY-NC-SA](https://creativecommons.org/licenses/by-nc-sa/4.0/) licensed
database.

In order to install the complete database you can check it out with:

```
$ pipenv install -e git+https://github.com/pyupio/safety-db.git#egg=safety-db
```

To use the local database, you have to enter the path to this database, in my case:

```
$ pipenv check --db /Users/veit/.local/share/virtualenvs/myproject-9TTuTZjx/src/safety-  

  ↪db/data
```

```

                /$$$$$          /$$
              /$$__ $$         | $$
 /$$$$$$ /$$$$$ | $$ \_//$$$$$ /$$$$$ /$$ /$$
/$$_____/ |____ $$| $$$ /$$__ $|_ $$_ / | $$ | $$
| $$$$$$ /$$$$$$| $$_ / | $$$$$$ | $$ | $$ | $$
 \_____ $$ /$$__ $$| $$ | $$$$$$ | $$ /$$| $$ | $$
 /$$$$$$$/| $$$$$$| $$ | $$$$$$ | $$$$/| $$$$$$
|_____/ \_____/|_ / \_____/ \____/ \_____ $$
                                     /$$ | $$
                                     | $$$$$$/
by pyup.io                         \_____/

```

```
REPORT
checked 21 packages, using local DB
```

```
No known security vulnerabilities found.
```

clean

`pipenv clean` uninstalls all packages not specified in `Pipfile.lock`.

graph

`pipenv graph` displays the dependency graph information for the currently installed packages.

install

`pipenv install` installs the provided packages and adds them to the pipfile. `pipenv install` knows the following options:

-d, --dev

installs the packages in `[dev-packages]`, for example:

```
$ pipenv install --dev pytest
...
$ cat Pipfile
...
[dev-packages]
pytest = "*"

```

--deploy

aborts if `Pipfile.lock` is out of date or an incorrect Python version is used.

-r, --requirements <requirements.txt>

imports a `requirements.txt` file.

--sequential

installs the dependency in a specific order, not at the same time.

While this slows down the installation, it increases the determinability of the builds.

sdist vs. wheel

`pip` can install packages as [Source Distribution \(sdist\)](#) or [Wheel](#). If both are present on [PyPI](#), `pip` will prefer a compatible [Wheel](#).

Note: However, dependencies on wheels are not covered by `$ pipenv lock`.

Requirement specifier

[Requirement specifier](#) specify the respective package.

- The latest version can be installed, for example:

```
$ pipenv install requests

```

- A specific version can be installed, for example:


```
$ pipenv install requests==2.18.4
```

- If the version has to be in a specific version range, this can also be specified:

```
$ pipenv install requests>=2,<3
```

- A compatible version can also be installed:

```
$ pipenv install requests~=2.18
```

This is compatible with ==2.18.*.

- For some packages, [optional dependencies](#) can also be specified with square brackets:

```
$ pipenv install requests[security]
```

- It can also be specified that certain packages are only installed on certain systems, so for the following Pipfile the module pywinusb is only installed on Windows systems.

```
[packages]
pywinusb = {version = "*", sys_platform = "=='win32'"}

```

A more complex example differentiates which module versions should be installed with which Python versions:

```
[packages]
unittest2 = {version = ">=1.0,<3.0", markers="python_version < '2.7.9' or (python_
↪version >= '3.0' and python_version < '3.4')"}

```

VCS

You can also install Python packages from version control, for example:

```
$ pipenv install -e git+https://github.com/requests/requests.git#egg=requests
```

Note: If `editable=false`, sub-dependencies are not resolved.

Further information on pipenv and VCS can be found in [Pipfile spec](#).

The version management credentials can also be specified in the pipfile, for example

```
[[source]]
url = "https://$USERNAME:$PASSWORD@pypi.cusy.io/simple"
verify_ssl = true
name = "cusy-pypi"

```

Note: pipenv hashes Pipfile before the environment variables are determined, and the environment variables are also written to Pipfile.lock, so that no credentials need to be stored in the version control.

lock

`pipenv lock` generates the file `Pipfile.lock` that lists all the dependencies and sub-dependencies of your project including the latest available versions and the current hash values for the downloaded files. This ensures repeatable and, above all, deterministic builds.

Note: In order to increase the determinism, the installation sequence can also be guaranteed in addition to the hash values. The `--sequential` flag is used for this.

Security features

`pipfile.lock` uses some security enhancements from `pip`: by default, sha256 hashes are generated for each downloaded package.

We strongly recommend `lock` using to deploy development environments to production. In the development environment you use `pipenv lock` to compile your dependencies and then you can use the compiled file `Pipfile.lock` in the production environment for reproducible builds.

open

`pipenv open MODULE` shows a specific module in your editor.

If you use [PyCharm](#), you have to configure `pipenv` for your Python project. How to do this is described in [Configuring Pipenv Environment](#).

run

`pipenv run` spawns a command that is installed in the virtual environment, for example:

```
$ pipenv run python main.py
```

shell

`pipenv shell` spawns a shell in the virtual environment. This gives you a Python interpreter that contains all Python packages and is therefore ideal for debugging and testing, for example:

```
$ pipenv shell --fancy
Launching subshell in virtual environment...
bash-4.3.30$ python
Python 3.6.4 (default, Jan 6 2018, 11:51:59)
>>> import requests
>>>
```

Note: Shells are usually not configured so that a subshell can be used. This can lead to unexpected results. In these cases `pipenv shell` should be used instead of `pipenv shell --fancy` as this uses a compatibility mode.

sync

`pipenv sync` installs all packages specified in `Pipfile.lock`.

uninstall

`pipenv uninstall` uninstalls all provided packages and removes them from the `Pipfile`. `uninstall` supports all parameters of *install* plus the following two options:

`--all`

deletes all files from the virtual environment, but leaves the `Pipfile` untouched.

`--all-dev`

removes all development packages from the virtual environment and removes them from the `Pipfile`.

update

`pipenv update` runs first `pipenv lock`, then `pipenv sync`.

`pipenv update` has the following options:

`--clear`

clears the *dependency cache*.

`--outdated`

lists obsolete dependencies.

Deterministic builds

All you have to do is specify what you want:

For example, `pipenv install requests` creates a `Pipfile` like the following:

```

[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
requests = "*"

[dev-packages]

[requires]
python_version = "3.6"

```

However, the associated `Pipfile.lock` file specifies the packages exactly, for example:

```

{
  "default": {
    "requests": {
      "hashes": [
        "sha256:63b52e3c866428a224f97cab011de738c36aec0185aa91cfacd418b5d58911d1

```

(continues on next page)

(continued from previous page)

```

    ↪",
        "sha256:ec22d826a36ed72a7358ff3fe56cbd4ba69dd7a6718ffd450ff0e9df7a47ce6a"
    ],
    "index": "pypi",
    "version": "==2.19.1"
},
"urllib3": {
    "hashes": [
    ↪",
        "sha256:a68ac5e15e76e7e5dd2b8f94007233e01effe3e50e8daddf69acfd81cb686baf
    ↪",
        "sha256:b5725a0bd4ba422ab0e66e89e030c806576753ea3ee08554382c14e685d117b5"
    ],
    "markers": "python_version != '3.2.*' and python_version != '3.1.*' and_
    ↪python_version < '4' and python_version != '3.3.*' and python_version >= '2.6' and_
    ↪python_version != '3.0.*'",
    "version": "==1.23"
    }
},
"develop": {}
}

```

Pipfile.lock also specifies all the dependencies of your project, whereby the hash values of the downloaded files are saved. This is to ensure repeatable and deterministic builds.

Workflows

Import and export of requirements.txt files

If you already have a requirements.txt file in an existing project, pipenv can resolve dependencies. If the requirements.txt file is in the same directory, simply with `$ pipenv install` or, if it is in a different directory, with `$ pipenv install -r /path/to/requirements.txt`.

Conversely, you can also create a requirements.txt file from an existing Pipenv environment with:

```
$ pipenv run pip freeze > requirements.txt
```

Upgrade workflow

1. Find out what has changed upstream:

```
$ pipenv update --outdated
Package 'requests' out-of-date: '==2.13.0' installed, '==2.19.1' available.
```

2. To update the Python packages, you have the following two options:

- update everything with `$ pipenv update`
- update individual packages, for example requests with `$ pipenv update requests`

Pipfile vs. setup.py

A distinction must be made whether you are developing an application or a library.

Libraries

They offer reusable functions for other libraries and applications/projects. You have to work with other libraries, each with their own dependencies. To avoid version conflicts in dependencies between different libraries within a project, libraries should never commit dependency versions. However, you can specify lower or upper limits if you are relying on a particular feature or bug fix. Library dependencies are noted in `install_requires` of the `setup.py` file.

Applications

They use libraries and are mostly not dependent on other projects. They should be implemented in a specific environment and only then should the exact versions of all their dependencies and sub-dependencies be specified. Facilitating this process is the main goal of Pipenv.

Environment variables

pipenv environment variables

`pipenv --envs` outputs options of the environment variables.

For more information, see [Configuration With Environment Variables](#).

.env file

If an `.env` file exists in your virtual environment, `$ pipenv shell` and `$ pipenv run` will automatically load it:

```
$ cat .env
USERNAME=veit

$ pipenv run python
Loading .env environment variables...
...
```

```
>>> import os
>>> os.environ["USERNAME"]
'veit'
```

The credentials of the version management, can also be specified in the Pipfile, for example:

```
[[source]]
url = "https://$USERNAME:${PASSWORD}@ce.cusy.io/api/v4/projects/$PROJECT_ID/packages/
↳ pypi/simple"
verify_ssl = true
name = "gitlab"
```

Note: pipenv hashes the pipfile before determining the environment variables, and the environment variables from the `pipfile.lock` are also replaced so that no credentials need to be stored in the version management.

You can also save the `.env` file outside your virtual environment. You then only have to specify the path to this file in `PIPENV_DOTENV_LOCATION`:

```
$ PIPENV_DOTENV_LOCATION=/PATH/TO/.env pipenv shell
```

You can also prevent pipenv from using an existing `.env` file with:

```
$ PIPENV_DONT_LOAD_ENV=1 pipenv shell
```

Pipenv and Spack

We need Pipenv for our *Spack environments* to be able to generate binary-compatible builds with Spack on the one hand and to be able to easily use Python packages for data collection, visualization, etc. on the other.

To do this, first activate the appropriate Python version from the Spack environment:

```
$ spack env activate python-311
$ spack env status
==> In environment python-311
$ which python
/srv/jupyter/spack/var/spack/environments/python-311/.spack-env/view/bin/python
```

Then you can install the existing Pipenv environment with:

```
$ cd ~/jupyter-tutorial/pipenvs/python-311/
$ pipenv --python=/Users/veit/jupyter-tutorial/spackenvs/python-311/.spack-env/
  ↳view/bin/python --site-packages
$ pipenv install
Creating a virtualenv for this project...
Pipfile: /Users/veit/jupyter-tutorial/pipenvs/python-311/Pipfile
Using /Users/veit/jupyter-tutorial/spackenvs/python-311/.spack-env/view/bin/
  ↳python3.11 (3.11.4) to create virtualenv...
...
```

This uses the environment installed with Spack and installs additional packages.

See also:

- [Pipenv and Other Python Distributions](#)

7.4 Creating programme libraries and packages

Learn how to create programme libraries and packages in our [Python Basics Tutorial](#).

7.5 Document

So that your product can be used effectively, documentation is required for the target groups of data scientists and data engineers as well as for system engineers:

- Data scientists want to see documented
 - which problems your product solves and what the main functions and limitations of the software are (README)
 - how the product can be used

- which changes have come in more recent software versions (CHANGELOG)
- Data engineers want to know how troubleshooting can help improve the product (CONTRIBUTING) and how they can communicate with others (CODE_OF_CONDUCT)
- System engineers need installation instructions for your product and the required dependencies

Together, they all need information about how the product is licensed (LICENSE file or LICENSES folder and how they can get help if needed).

See also:

- [Document](#)
- [Read the Docs for Science](#)

7.6 Licensing

In order for others to use your software, it should have one or more licences that describe the terms of use. Otherwise, it is likely to be protected by copyright. Authors are those who have originally contributed to the software. If software is to be licensed, the consent of all those who can claim authorship is required.

Note: This does not constitute legal advice. If in doubt, contact a lawyer or the legal department of your company.

See also:

- [The Whys and Hows of Licensing Scientific Code](#)
- [A Quick Guide to Software Licensing for the Scientist-Programmer](#)
- [Karl Fogel: Producing Open Source Software](#)
- [Forschungsdaten veröffentlichen](#)

7.6.1 Proprietary software licenses

Proprietary software licenses are rarely standardised; they can be commercial, shareware, or freeware.

7.6.2 Free and open source software licenses

They are defined by the [Free Software Foundation \(FSF\)](#) and the [Open Source Initiative \(OSI\)](#). A distinction can essentially be made between copyleft, permissive and public domain licenses.

Copyleft or reciprocal licences

Copyleft licences oblige the licensees to place any adaptation of the software (so-called derivatives) under the licence of the original work. This is intended to prevent restrictions on the use of the software. The best-known copyleft licence is the [GPL \(GNU General Public License\)](#). The copyleft of the GPL (GNU General Public License) is considered very strong, while that of the [Mozilla Public License](#) is considered very weak.

Since the licensors are not bound by their own copyleft, they can also publish new versions under a proprietary license or allow third parties to do so (multiple licensing).

Copyleft licenses can quickly lead to incompatibilities with free licenses without copyleft. For example, the 3 Clause BSD license is incompatible with the

However, copyleft licences can quickly create incompatibilities when distributed together with software under other free licences. For example, the 3-Clause BSD licence is incompatible with the GPL.

The [EUPL](#), on the other hand, is a reciprocal licence that is at least compatible and interoperable with most other open reciprocal licences: the compatible licence obligations take precedence if they conflict with the obligations arising from the EUPL.

Permissive open source licenses

Permissive open source licenses allow broader reuse than copyleft licenses. Derivatives and copies of the source code can be distributed under conditions that have fundamentally different properties than those of the original license. The best known examples of such licenses are [MIT](#) and [BSD](#).

Public domain licenses

With public domain licences, the copyrights are transferred to the general public. The [WTFPL](#) was created to mark the public domain of software.

7.6.3 Non-software licences

Open source software licences can also be used for works that are not software. They are often also the best choice, especially if the works in question are edited and versioned as source code.

Data, media, etc.

[CC0 1.0](#), [CC BY 4.0](#) and [CC BY-SA 4.0](#) are open licences used for non-software material, from datasets to videos. However, they are [not recommended for software](#).

The [Open Knowledge Foundation](#) has also published a set of [Open Data Commons](#) licences for data/databases:

Open Data Commons Open Database License (ODbL) v1.0

Attribution and sharing under equal terms.

Open Data Commons Attribution License (ODC-By) v1.0

Attribution.

Open Data Commons Public Domain Dedication and License (PDDL) v1.0

The PDDL places the data in the public domain and waives all rights.

GovData has submitted the *Data Licence Germany* in two variants:

- [Datenlizenz Deutschland – Namensnennung – Version 2.0](#)
- [Datenlizenz Deutschland – Zero – Version 2.0](#)

When using the [Community Data License Agreement – Permissive, Version 2.0](#) the copyright notices must be retained.

Another possible licence for artistic works is the [Free Art License 1.3](#).

Documentation

Any open source software licence or open media licence also applies to software documentation. If you use different licences for your software and its documentation, you should make sure that the source code examples in the documentation are also licensed under the software licence. In addition to the Creative Commons licences mentioned above, the following licences are available specifically for free documentation.

GNU Free Documentation License (FDL)

Copyright licence for documentation to be used for all GNU manuals. Its applicability is limited to textual works (books).

FreeBSD Documentation License

Permissive documentation licence with copyleft, compatible with the GNU FDL.

Open Publication License, Version 1.0

Free documentation licence with copyleft, provided none of the licence options in Section VI of the licence are used. In any case, it is incompatible with the GNU FDL.

Fonts

SIL Open Font License 1.1

Font licence that can be freely used in other works.

GNU General Public License 3

It can also be used for fonts, but it may only be included in documents with the [font exception](#).

See also:

- [Font Licensing](#)

LaTeX ec fonts

Free *European Computer Modern and Text Companion* fonts commonly used with LaTeX.

Arphic Public License

Free licence with copyleft.

IPA Font license

Free licence with copyleft, but derived values may not use or contain the name of the original.

Hardware

Designs for [open source hardware](#) are covered by the CERN Open Hardware licences:

CERN-OHL-P-2.0

permissive

CERN-OHL-W-2.0

weakly reciprocal

CERN-OHL-S-2.0

strongly reciprocal

7.6.4 Choosing a suitable license

Overviews of possible licenses can be found in the [SPDX License List](#) or [OSI Open Source Licenses by Category](#). When choosing suitable licences, the websites [Choose an open source license](#) and [Comparison of free and open-source software licenses](#) will help you.

If you want to achieve the widest possible distribution of your package, for example, MIT or BSD versions are a good choice. The Apache licence protects you better from patent infringement, but it is not compatible with the GPL v2.

Check dependencies

In addition, you should look at what licences those packages have that you depend on and should be compatible with:

Fig. 5: Licence compatibility for derivative works or combined works of own code and external code licensed under an open source licence (from [Licence compatibility](#), following [The Rise of Open Source Licensing](#) p. 119).

To analyse licences, you can look at [license compatibility](#).

With `liccheck` you can check Python packages and their dependencies with a `requirements.txt` file, for example:

```
liccheck -s liccheck.ini -r requirements.txt
gathering licenses...
3 packages and dependencies.
check unknown packages...
3 packages.
  cffi (1.15.1): ['MIT']
    dependency:
      cffi << cryptography
  cryptography (41.0.3): ['Apache Software', 'BSD']
    dependency:
      cryptography
  pycparser (2.21): ['BSD']
    dependency:
      pycparser << cffi << cryptography
```

Furthermore, it can also be useful to publish a package under several licences. An example of this is [cryptography/LICENSE](#):

This software is made available under the terms of *either* of the licenses found in LICENSE.APACHE or LICENSE.BSD. Contributions to cryptography are made under the terms of *both* these licenses.

The code used in the OpenSSL locking callback and OS random engine is derived from the same in CPython, and is licensed under the terms of the PSF License Agreement.

7.6.5 GitHub

On [GitHub](#) you can have an open source license created in your repository.

1. Go to the main page of your repository.
2. Click on *Create new file* and then enter `LICENSE` or `LICENSE.md` as the file name.
3. Then you can click on *Choose a license template*.
4. Now you can select the open source license that is suitable for your repository.
5. You will now be asked for additional information if the selected license requires this.
6. After you have given a commit message, for example `Add license`, you can click on *Commit new file*.

If you've already added a `/LICENSE` file to your repository, GitHub uses [licensee](#) to compare the file with a short list of open source licenses. If GitHub can't detect your repository's license, it might contain multiple licenses or be too complex. Then consider whether you can simplify the license, for example by outsourcing complexity to the `/README` file.

Conversely, you can also search for repositories with specific licenses or license families on GitHub. You can get an overview of the license keywords in [Searching GitHub by license type](#).

Finally, you can have [Shields.io](#) generate a license badge for you, which you can include in your `README` file, for example

```
|License|

.. |License| image:: https://img.shields.io/github/license/veit/python4datascience.svg
   :target: https://github.com/veit/python4datascience/blob/main/LICENSE
```

7.6.6 Standard format for licensing

SPDX stands for *Software Package Data Exchange* and defines a standardised method for the exchange of copyright and licensing information between projects and people. You can choose the appropriate SPDX identifiers from the [SPDX License List](#) and then add to the header of your licence files:

```
# SPDX-FileCopyrightText: [year] [copyright holder] <[email address]>
#
# SPDX-License-Identifier: [identifier]
```

7.6.7 Check conformity

REUSE

REUSE was initiated by the Free Software Foundation Europe (FSFE) to facilitate the licensing of free software projects. The [REUSE tool](#) checks licenses and supports you in compliance with the license, for example:

```
reuse lint
# MISSING COPYRIGHT AND LICENSING INFORMATION

The following files have no copyright and licensing information:
* .gitattributes
* .github/ISSUE_TEMPLATE/openssl-release.md
```

(continues on next page)

(continued from previous page)

```
...
* vectors/cryptography_vectors/x509/wosign-bc-invalid.pem
* vectors/pyproject.toml
```

The following files have no licensing information:

```
* docs/_ext/linkcode_res.py
* src/cryptography/__about__.py
```

SUMMARY

```
* Bad licenses: 0
* Deprecated licenses: 0
* Licenses without file extension: 0
* Missing licenses: 0
* Unused licenses: 0
* Used licenses: 0
* Read errors: 0
* files with copyright information: 2 / 2806
* files with license information: 0 / 2806
```

Unfortunately, your project is not compliant with version 3.0 of the REUSE Specification. ↪ :- (

With the [REUSE API](#) you can also generate a dynamic compliance badge:

CI workflow

You can easily integrate REUSE into your continuous integration workflow:

You can automatically run `reuse lint` as a *pre-commit hook* on every commit by adding the following to your `.pre-commit-config.yaml`:

```
repos:
- repo: https://github.com/fsfe/reuse-tool
  rev: v2.1.0
  hooks:
  - id: reuse
```

Add the following to the `.gitlab-ci.yml` file:

```
reuse:
  image:
    name: fsfe/reuse:latest
    entrypoint: [""]
  script:
  - reuse lint
```

On GitHub you can integrate the REUSE action into your workflow with the GitHub Action [REUSE Compliance Check](#), for example, by adding the following to your workflow `.yml` file:

```

name: REUSE Compliance Check

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: REUSE Compliance Check
        uses: fsfe/reuse-action@v2

```

Alternatives

ISO/IEC 5230/OpenChain

recommends *REUSE* as a component to improve license and copyright clarity, but sets higher requirements to achieve full compliance.

It is based on [OpenChain Specification 2.1](#) and is an international standard on software supply chains, simplified procurement, and open source license compliance.

See also:

- [OpenChain project](#)
- [OpenChain Self Certification](#)
- [Reference-Material](#)

ScanCode

offers a range of tools and applications for scanning software codebases and packages to determine the origin and licence (provenance) of open source software (and other third-party software).

DeltaCode

compares two codebase scans to detect significant changes.

ClearlyDefined

collects and displays information about the licensing and copyright situation of a software project.

FOSSology

is a free software compliance toolkit that stores information in a database with license, copyright, and export scanners.

OSS Review Toolkit (ORT)

is a toolkit for automating and orchestrating FOSS policies, allowing you to manage your (open source) software dependencies. It

- generates [OWASP CycloneDX](#), [SPDX Software Bill of Materials \(SBOM\)](#) or custom FOSS attribution documentation for your software project
- automates your FOSS policy to check your software project and its dependencies for licensing, security vulnerabilities, source code and technical standards
- create a source code archive for your software project and its dependencies to comply with specific licenses
- correct package metadata or license findings yourself

See also:

- [GitHub Action for ORT](#)

Search for descriptors like "composer", or "gem"

Workspace Revert Changes

Component

✓ cryptography / 41.0.3
(Apache-2.0 OR (Apache-2.0 AND BSD-3-Clause))

Declared: Apache-2.0 OR (Apache-2.0 AND BSD-3-Clause) Discovered: Apache-2.0, Apache-2.0 AND BSD-3-Clause, A...

Source: <https://pypi.org/project/cryptography/41.0.3/> Attribution: Copyright 2013-2023, copyright 2013-2023, In...

Release: 2023-08-01 Files: Total: 375 | Licensed: 288 (77%) | Attributed: 4

<https://clearlydefined.io/workspace#>

Overall

Effective: 87/100 Tools: 87/100

Described

Effective: 100/100 Tools: 100/100

Date: 30/30 Date: 30/30

Source: 70/70 Source: 70/70

Licensed

Effective: 75/100 Tools: 75/100

Consistency: 15/15 Consistency: 15/15

Declared: 30/30 Declared: 30/30

Discovered: 0/25 Discovered: 0/25

SPDX: 15/15 SPDX: 15/15

License texts: 15/15 License texts: 15/15

[Scoring Formula](#)

- [ORT for GitLab](#)

licensechecker

A command line tool that scans installation directories for licences.

7.6.8 Python package metadata

With [PEP 658](#) the METADATA file from distributions becomes available in the [PEP 503](#) repository API on [PyPI](#). This allows the metadata of [distribution packages](#) to be analysed without having to download the whole package.

In Python packages there are other fields where licence information is stored, such as the [core metadata specifications](#), which are also limited. This leads not only to problems for authors to specify the correct licence, but also to problems when re-packaging for various Linux distributions.

Currently, although some common cases are covered and the licence classification can also be extended, there are some popular classifications such as `License :: OSI Approved :: BSD License` that will be abolished. However, this means that backwards compatibility is no longer guaranteed and the packages have to be relicensed. At least you have a way to check your trove classifications with [trove-classifiers](#).

See also:

- [PEP 639](#) – Improving License Clarity with Better Package Metadata
- [PEP 621](#) – Storing project metadata in `pyproject.toml`
- [PEP 643](#) – Metadata for Package Source Distributions

7.7 Citing

Today software and data are integral parts of scientific research. Software is used to create, process and analyse research data and to model and simulate complex processes. Despite their increasing importance in research, it's little known how they can be embedded in the scientific recognition and reputation systems. Quotations are an essential option in these systems, but few researchers know how software and data could be cited.

Unfortunately, there are no recognised guidelines for software and data authorship. In addition to the *programmers* role, other roles such as *software architects*, *technical writers* and *maintainers* can also be defined.

See also:

- ICMJE: Defining the Role of Authors and Contributors
- Bot Recognize All Contributors

7.7.1 Cite data

DataCite Metadata Schema

The DataCite Metadata Working Group published the DataCite Metadata Schema Documentation for the publication and citation of research data in 2019: [DataCite Metadata Schema 4.3](#) together with a XSD (XML Schema Definition): [metadata.xsd](#).

A simple datacite example can look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<resource xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://datacite.
↪org/schema/kernel-4" xsi:schemaLocation="http://datacite.org/schema/kernel-4 http://
↪schema.datacite.org/meta/kernel-4.3/metadata.xsd">
  <identifier identifierType="DOI">10.5072/D3P26Q35R-Test</identifier>
  <creators>
    <creator>
      <creatorName nameType="Personal">Fosmire, Michael</creatorName>
      <givenName>Michael</givenName>
      <familyName>Fosmire</familyName>
    </creator>
    <creator>
      <creatorName nameType="Personal">Wertz, Ruth</creatorName>
      <givenName>Ruth</givenName>
      <familyName>Wertz</familyName>
    </creator>
    <creator>
      <creatorName nameType="Personal">Purzer, Senay</creatorName>
      <givenName>Senay</givenName>
      <familyName>Purzer</familyName>
    </creator>
  </creators>
  <titles>
    <title xml:lang="en">Critical Engineering Literacy Test (CELT)</title>
  </titles>
  <publisher xml:lang="en">Purdue University Research Repository (PURR)</publisher>
  <publicationYear>2013</publicationYear>
  <subjects>
```

(continues on next page)

(continued from previous page)

```

<subject xml:lang="en">Assessment</subject>
<subject xml:lang="en">Information Literacy</subject>
<subject xml:lang="en">Engineering</subject>
<subject xml:lang="en">Undergraduate Students</subject>
<subject xml:lang="en">CELT</subject>
<subject xml:lang="en">Purdue University</subject>
</subjects>
<language>en</language>
<resourceType resourceTypeGeneral="Dataset">Dataset</resourceType>
<version>1.0</version>
<descriptions>
  <description xml:lang="en" descriptionType="Abstract">
    We developed an instrument, Critical Engineering Literacy Test (CELT), which is a
    ↪ multiple choice instrument designed to measure undergraduate students' scientific and
    ↪ information literacy skills. It requires students to first read a technical memo
    and, based on the memo's arguments, answer eight multiple choice and six open-
    ↪ ended response questions. We collected data from 143 first-year engineering students
    ↪ and conducted an item analysis. The KR-20 reliability of the instrument was .39. Item
    difficulties ranged between .17 to .83. The results indicate low reliability index
    ↪ but acceptable levels of item difficulties and item discrimination indices. Students
    ↪ were most challenged when answering items measuring scientific and mathematical
    literacy (i.e., identifying incorrect information).
  </description>
</descriptions>
</resource>

```

W3C-PROV

The PROV document family of the W3C working group defines various aspects that are necessary to be able to exchange provenance information interoperably.

See also:

- [Provenance: An Introduction to PROV](#) by Luc Moreau and Paul Groth
- [Provenance storage and distribution](#)
- [ProvStore's API documentation](#)

Python prov

With [prov](#), a Python3 library is available that supports the import and export of the [PROV data model](#) into the following serialisation formats:

- [PROV-O \(RDF\)](#)
- [PROV-XML](#)
- [PROV-JSON](#)

In addition, PROV documents can be created with [NetworkX MultiDiGraph](#) and vice versa. Finally, PROV documents can also be generated as graphs in PDF, PNG and SVG formats.

See also:

- [A Short Tutorial for Prov Python by Dong Huynh](#)
- [PROV Tutorial.ipynb](#)

7.7.2 Cite software

James Howison and Julia Bullard listed the following examples in descending reputations in their 2016 article [Software in the scientific literature](#):

1. citing publications that describe the respective software
2. citing operating instructions
3. citing the software project website
4. link to a software project website
5. mention the software name

The situation remains unsatisfactory for the authors of software, especially if they differ from the authors of the software description. Conversely, research software is unfortunately not always well suited to being cited. For example, others will hardly be able to cite your software directly if you send it to them as an email attachment. Even a download link is not really useful here. It is better to provide a [persistent identifier \(PID\)](#) to ensure the long-term availability of your software. Both [Zenodo](#) and [figshare](#) repositories accept source code including binaries and provide [Digital Object Identifiers \(DOI\)](#) for them. The same applies to [CiteAs](#), which can be used to retrieve citation information for software.

See also:

- [Should I cite?](#)
- [How to cite software “correctly”](#)
- [Daniel S. Katz: Compact identifiers for software: The last missing link in user-oriented software citation?](#)
- [Neil Chue Hong: How to cite software: current best practice](#)
- [Recognizing the value of software: a software citation guide](#)
- [Stephan Druskat, Radovan Bast, Neil Chue Hong, Alexander Konovalov, Andrew Rowley, Raniere Silva: A standard format for CITATION files](#)
- [Module-5-Open-Research-Software-and-Open-Source](#)
- [Software Heritage: Save and reference research software](#)
- [Mining software metadata for 80 M projects and even more](#)
- [Extensions to schema.org to support structured, semantic, and executable documents](#)
- [Guide to Citation File Format schema](#)
- [schema.json](#)

Create a DOI with Zenodo

Zenodo enables software to be archived and a DOI to be provided for it. In the following I will show which steps are required on the example of the Jupyter tutorial:

1. If you haven't already, [create an account on Zenodo](#), preferably with GitHub.
2. In *Upload* → *New Upload* under *Basic information* activate the button *Reserve DOI* to reserve a DOI (Digital Object Identifier) for your upload. Leave the form open to upload your software later.
3. Create or modify the *CodeMeta*- und *Citation File Format* files in your software directory.
4. Include the badge in the README file of your software:

Markdown:

```
[![DOI](https://zenodo.org/badge/307380211.svg)](https://zenodo.org/badge/latestdoi/307380211)
```

reStructuredText:

```
.. image:: https://zenodo.org/badge/307380211.svg
   :target: https://zenodo.org/badge/latestdoi/307380211
```

5. Now select the repository that you want to archive:

6. Check whether Zenodo has created a webhook in your repository for the *Releases* event:
7. Create a new release:

veit / jupyter-tutorial

Unwatch3

Star4

Fork1

<> Code

Issues37

Pull requests

Actions

Projects

Wiki

Security

Insights

Settings

Options

Manage access

Security & analysis

Branches

Webhooks

Notifications

Integrations

Deploy keys

Secrets

Actions

Moderation settings

Interaction limits

Webhooks

Add webhook

Webhooks allow external services to be notified when certain events happen. When the specified events happen, we'll send a POST request to each of the URLs you provide. Learn more in our [Webhooks Guide](#).

✓ <https://readthedocs.org/api/v2/...> (create, delete, pull_requ...)

EditDelete

✓ <https://pyup.io/provider/github/...> (push)

EditDelete

✓ <https://zenodo.org/api/hooks/r...> (release)

EditDelete

veit / jupyter-tutorial

Unwatch3

Star4

Fork1

<> Code

Issues37

Pull requests

Actions

Projects

Wiki

Security

Insights

Settings

Releases

Tags

Draft a new release

Latest release

0.8.0

5175154

Compare

0.8.0

veit released this 1 hour ago · 3 commits to master since this release

- Switch to English documentation
- Minor fixes for bugs and typos
- Add NoSQL databases
- Add «What's new in Python 3.9?»

Assets2

Source code (zip)

Source code (tar.gz)

0.7.0

on 2 Aug

Metadata formats

The [FORCE11](#) working group has published a paper in which the principles of scientific software citation are presented: [FORCE11 Software Citation Working Group](#) by Arfon Smith, Daniel Katz and Kyle Niemeyer 2016. Two projects are currently emerging for structured metadata:

CodeMeta

is an exchange scheme for general software metadata and reference implementation for JSON for Linking Data (JSON-LD).

Citation File Format

is a scheme for software citation metadata in machine-readable [YAML](#) format.

CodeMeta

[CodeMeta](#) is an exchange scheme for general software metadata and reference implementation for JSON for Linking Data (JSON-LD).

A `codemeta.json` file is expected in the root directory of the software repository. The file can look like this:

```
{
  "@context": "https://doi.org/10.5063/schema/codemeta-2.0",
  "@type": "SoftwareSourceCode",
  "author": [{
    "@type": "Person",
    "givenName": "Stephan",
    "familyName": "Druskat",
    "@id": "http://orcid.org/0000-0003-4925-7248"
  }],
  "name": "My Research Tool",
  "softwareVersion": "2.0",
  "identifier": "https://doi.org/10.5281/zenodo.1234",
  "datePublished": "2017-12-18",
  "codeRepository": "https://github.com/research-software/my-research-tool"
}
```

See also:

- [CodeMeta generator](#)
- [Codemeta Terms](#)
- [GitHub Repository](#)

Citation File Format

[Citation File Format](#) is a scheme for software citation metadata in machine-readable [YAML](#) format.

A file `CITATION.cff` should be stored in the root directory of the software repository.

The content of the file can look like this:

```
cff-version: "1.1.0"
message: "If you use this tutorial, please cite it as below."
authors:
```

(continues on next page)

(continued from previous page)

```

-
  family-names: Schiele
  given-names: Veit
  orcid: "https://orcid.org/https://orcid.org/0000-0002-2448-8958"
identifiers:
-
  type: doi
  value: "10.5281/zenodo.4147287"
keywords:
- "data-science"
- jupyter
- "jupyter-notebooks"
- "jupyter-kernels"
- ipython
- pandas
- spack
- pipenv
- ipywidgets
- "ipython-widget"
- dvc
title: "Jupyter tutorial"
version: "0.8.0"
date-released: 2020-10-08
license: "BSD-3-Clause"
repository-code: "https://github.com/veit/jupyter-tutorial"

```

You can easily adapt the example above to create your own CITATION.cff file or use the [cffinit](#) website.

With [cff-validator](#) you have a GitHub action that checks CITATION.cff files with the R package V8.

Tools

Git2PROV

generates PROV data from the information in a Git repository. generiert PROV-Daten aus den Informationen eines Git-Repository.

HERMES

simplifies the publication of research software by continuously retrieving existing metadata in *Citation File Format*, *CodeMeta* and *Git*.

Git2PROV

[Git2PROV](#) generates PROV data from the information in a Git repository.

On the command line, the conversion can be easily executed with:

```
$ git2prov git_url [serialization]
```

For example:

```
$ git2prov git@github.com:veit/python4datascience.git PROV-JSON
```

In total, the following serialisation formats are available:

- PROV-N
- PROV-JSON
- PROV-O
- PROV-XML

Alternatively, Git2PROV also provides a web server with:

```
$ git2prov-server [port]
```

See also:

- [Git2PROV: Exposing Version Control System Content as W3C PROV](#)
- [GitHub-Repository](#)

HERMES

HERMES simplifies the publication of research software by continuously retrieving existing metadata in *Citation File Format*, *CodeMeta* and *Git*. Subsequently, the metadata is also compiled appropriately for *InvenioRDM* and *Dataverse*. Finally, *CITATION.cff* and *codemeta.json* are also updated for the publication repositories.

1. Add `.hermes/` to the `.gitignore` file
2. Provide *CITATION.cff* file with additional metadata

Important: Make sure `license` is defined in the *CITATION.cff* file; otherwise, your release will not be accepted as open access by the *Zenodo* sandbox.

3. Configure HERMES workflow

The HERMES workflow is configured in the file *TOML*, where each step gets its own section.

If you want to configure HERMES to use the metadata from *Git* and *CITATION.cff*, and to file in the Zenodo sandbox built on InvenioRDM, the `hermes.toml` file looks like this:

Listing 2: hermes.toml

```
# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

[harvest]
from = [ "git", "cff" ]

[deposit]
mapping = "invenio"
target = "invenio"

[deposit.invenio]
site_url = "https://sandbox.zenodo.org"
access_right = "open"

[postprocess]
execute = [ "config_record_id" ]
```

4. Access token for Zenodo Sandbox

In order for GitHub Actions to publish your repository in the [Zenodo Sandbox](#), you need a personal access token. To do this, you need to log in to Zenodo Sandbox and then create a [personal access token](#) in your user profile with the name `HERMES workflow` and the scopes `deposit:actions` und `deposit:write`:

New personal access token

Name

HERMES workflow

Name of personal access token.

Scopes

☒ **deposit:actions**
Allow publishing of uploads.

☒ **deposit:write**
Allow upload (but not publishing).

☐ **user:email**
Allow access to email address (read-only).

Scopes assign permissions to your personal access token. A personal access token works just like a normal OAuth access token for authentication against the API.

- Copy the newly created token to a new [GitHub secret](#) named `ZENODO_SANDBOX` in your repository: *Settings* → *Secrets and Variables* → *Actions* → *New repository secret*:

General

Access

Collaborators

Moderation options

Code and automation

Branches

Tags

Rules Beta

Actions

Webhooks

Environments

Codespaces

Pages

Security

Code security and analysis

Deploy keys

Secrets and variables

Actions

Codespaces

Dependabot

Integrations

GitHub Apps

Email notifications

Actions secrets / New secret

Name *

ZENODO_SANDBOX

Secret *

Add secret

6. Configure the GitHub action

The HERMES project provides templates for continuous integration in a special repository: [hermes-hmc/ci-templates](#). Copy the template file `TEMPLATE_hermes_github_to_zenodo.yml` into the `.github/workflows/` directory of your repository and rename it, for example to `hermes_github_to_zenodo.yml`.

Then you should go through the file and look for comments marked `# ADAPT`. Modify the file to suit your needs.

Finally, add the workflow file to version control and push it to the GitHub server:

```
$ git add .github/workflows/hermes_github_to_zenodo.yml
$ git commit -m ":construction_worker: GitHub action for automatic publication with_
->HERMES"
$ git push
```

7. GitHub actions should be allowed to create pull requests in your repository

The HERMES workflow will not publish metadata without your approval. Instead, it will create a pull request so that you can approve or change the metadata that is stored. To enable this, go to *Settings* → *Actions* → *General* in your repository and in the *Workflow permissions* section, enable *Allow GitHub Actions to create and approve pull requests*.

7.7.3 Software journals

General

- IEEE Computer Society Digital Library
- Wiley Online Library
- Journal of Open Source Software
- Journal of Open Research Software (JORS)
- Journal of Software: Practice and Experience
- Nature Toolbox
- Research Ideas and Outcomes (RIO)
- SIAM Journal on Scientific Computing (SISC) Software section
- SoftwareX

Image processing

- Image Processing On Line
- Insight Journal

Biology

- American Journal of Human Genetics
- Artificial Life
- Psychonomic Society: Behaviour Research Methods
- Oxford Academic: Bioinformatics
- Bioinformatics and Biology Insights
- Biophysical Journal
- BMC Bioinformatics
- BMC Systems Biology
- Bone

- Computer Methods and Programs in Biomedicine
- Current Protocols in Bioinformatics
- Database: The Journal of Biological Databases and Curation
- Ecography
- eLife
- Epidemiology
- Evolutionary Bioinformatics
- F1000 Research
- Frontiers in Neuroinformatics
- Gigascience
- Methods in Ecology and Evolution
- Nature Methods
- Neurocomputing
- Neuroinformatics
- Nucleic Acids Research
- PeerJ - Life and Environment
- PLoS Computational Biology: Software collection
- PLoS ONE
- Trends in Parasitology

Chemistry

- International Journal of Quantum Chemistry
- Journal of Applied Crystallography
- Journal of Chemical Theory and Computation
- Journal of Chemical Information and Modelling
- Journal of Cheminformatics
- Journal of Computational Chemistry
- Molecular Simulation
- Wiley Interdisciplinary Reviews: Computational Molecular Science

Human and social sciences

- Digital Humanities Quarterly
- Journal of Artificial Societies and Social Simulation
- Journal of Economic Dynamics and Control

Engineering

- Advances in Engineering Software
- Coastal Engineering
- Renewable Energy

Computer science, mathematics and statistics

- ACM Transactions on Mathematical Software
- The Archive of Numerical Software
- Future Generation Computer Systems
- Journal of Machine Learning Research: Machine Learning Open Source Software track
- Journal of Multiscale Modelling and Simulation
- Journal of Parallel and Distributed Computing
- Journal of Software for Algebra and Geometry
- Journal of Statistical Software
- Knowledge-Based Systems
- LMS Journal of Computation and Mathematics
- The Mathematica Journal
- Mathematical Programming Computation
- Numerical Algorithms
- PeerJ Computer Science
- The R Journal
- Science of Computer Programming
- The Stata Journal

Physics and Earth Sciences

- AAS: The Astronomy Journal
- AAS: The Astrophysical Journal
- AAS: The Astrophysical Journal Supplement Series
- Astronomy and Computing
- Communications in Computational Physics

- Computational Astrophysics and Cosmology
- Computer Physics Communications
- Computers and Geosciences
- Computing and Software for Big Science
- Environmental Modelling & Software
- Geoscientific Model Development

7.8 Testing

All the options you have for [testing your notebooks](#) are also available for Python packages. In addition, you can also check the test coverage of your package and have your tests executed automatically on a regular basis.

See also:

[Testing](#)

7.9 Logging

The [logging](#) module is part of the Python standard library. It is described in [PEP 0282](#). You can get a first introduction to the module in the [Basic Logging Tutorial](#).

Logging usually serves two different purposes:

- **Diagnosis:**
 - You can display the context of certain events.
 - Tools like [Sentry](#) group related events and facilitate user identification, etc., so that developers can find the cause of the error more quickly.
- **Monitoring:**
 - The logging records events for user-defined heuristics, for example for business analyses. These records can be used for reports or optimisation of the business goals and, if necessary, visualised.

What are the advantages of [logging](#) over [print](#)?

- The log file contains all available diagnostic information such as file name, path, function and line number.
- All events are automatically available via the root logger unless they are explicitly filtered out.
- Logging can be muted using either of the following two methods: [logging.Logger.setLevel\(\)](#) or [logging.disabled](#).

See also:

- [loguru](#), which makes logging almost as easy as using print instructions.
- [structlog](#) adds structure to your log entries.

7.9.1 Logging examples

Creating a log file

```
[1]: import logging

logging.warning("This is a warning message")
logging.critical("This is a critical message")
logging.debug("debug")

WARNING:root:This is a warning message
CRITICAL:root:This is a critical message
```

Logging levels

Level	Description
CRITICAL	The programme was stopped
ERROR	A serious error has occurred
WARNING	An indication that something unexpected has happened (default level)
INFO	Confirmation that things are working as expected
DEBUG	Detailed information that is usually only of interest when diagnosing problems

Setting the logging level

```
[2]: import logging

logging.basicConfig(filename="example.log", filemode="w", level=logging.INFO)

logging.info("Informational message")
logging.error("An error has happened!")

ERROR:root:An error has happened!
```

Creating a Logger Object

```
[3]: import logging

logging.basicConfig(filename="example.log")
logger = logging.getLogger("example")
logger.setLevel(logging.INFO)

try:
    raise RuntimeError
except Exception:
    logger.exception("Error!")
```

```
ERROR:example:Error!
Traceback (most recent call last):
  File "/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_65916/2646645271.py", line 9, in <module>
    raise RuntimeError
RuntimeError
```

Logging exceptions

```
[4]: try:
      1 / 0
except ZeroDivisionError:
    logger.exception("You can't do that!")
```

```
ERROR:example:You can't do that!
Traceback (most recent call last):
  File "/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_65916/760044062.py", line 2, in <module>
    1 / 0
    ~^^~
ZeroDivisionError: division by zero
```

Logging handler

Handler types

Handler	Description
StreamHandler	stdout, stderr or file-like objects
FileHandler	for writing to disk
RotatingFileHandler	supports log rotation
TimedRotatingFileHandler	supports the rotation of log files on the hard disk at specific time intervals
SocketHandler	sends logging output to a network socket
SMTPHandler	supports sending logging messages to an e-mail address via SMTP

See also

Further handlers can be found at [Logging handlers](#)

StreamHandler

```
[5]: import logging

logger = logging.getLogger("stream_logger")
logger.setLevel(logging.INFO)

console = logging.StreamHandler()
```

(continues on next page)

(continued from previous page)

```
logger.addHandler(console)
logger.info("This is an informational message")
```

This is an informational message
INFO:stream_logger:This is an informational message

SMTPHandler

```
[6]: import logging
import logging.handlers

logger = logging.getLogger("email_logger")
logger.setLevel(logging.INFO)
fh = logging.handlers.SMTPHandler(
    "localhost",
    fromaddr="python-log@localhost",
    toaddrs=["logs@cusy.io"],
    subject="Python log",
)
logger.addHandler(fh)
logger.info("This is an informational message")
```

```
--- Logging error ---
Traceback (most recent call last):
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↳ 11/lib/python3.11/logging/handlers.py", line 1081, in emit
    smtp = smtplib.SMTP(self.mailhost, port, timeout=self.timeout)
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↳ 11/lib/python3.11/smtplib.py", line 255, in __init__
    (code, msg) = self.connect(host, port)
                  ^^^^^^^^^^^^^^^^^^^^^
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↳ 11/lib/python3.11/smtplib.py", line 341, in connect
    self.sock = self._get_socket(host, port, self.timeout)
                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↳ 11/lib/python3.11/smtplib.py", line 312, in _get_socket
    return socket.create_connection((host, port), timeout,
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↳ 11/lib/python3.11/socket.py", line 851, in create_connection
    raise exceptions[0]
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↳ 11/lib/python3.11/socket.py", line 836, in create_connection
    sock.connect(sa)
ConnectionRefusedError: [Errno 61] Connection refused
Call stack:
  File "<frozen runpy>", line 198, in _run_module_as_main
```

(continues on next page)

(continued from previous page)

```

File "<frozen runpy>", line 88, in _run_code
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/ipykernel_launcher.py", line 17, in <module>
    app.launch_new_instance()
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/traitlets/config/application.py", line 1043, in launch_instance
    app.start()
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/ipykernel/kernelapp.py", line 736, in start
    self.io_loop.start()
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/tornado/platform/asyncio.py", line 195, in start
    self.asyncio_loop.run_forever()
File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↳ 11/lib/python3.11/asyncio/base_events.py", line 607, in run_forever
    self._run_once()
File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↳ 11/lib/python3.11/asyncio/base_events.py", line 1922, in _run_once
    handle._run()
File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↳ 11/lib/python3.11/asyncio/events.py", line 80, in _run
    self._context.run(self._callback, *self._args)
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/ipykernel/kernelbase.py", line 516, in dispatch_queue
    await self.process_one()
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/ipykernel/kernelbase.py", line 505, in process_one
    await dispatch(*args)
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/ipykernel/kernelbase.py", line 412, in dispatch_shell
    await result
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/ipykernel/kernelbase.py", line 740, in execute_request
    reply_content = await reply_content
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/ipykernel/ipkernel.py", line 422, in do_execute
    res = shell.run_cell(
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/ipykernel/zmqshell.py", line 546, in run_cell
    return super().run_cell(*args, **kwargs)
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/IPython/core/interactiveshell.py", line 3009, in run_cell
    result = self._run_cell(
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/IPython/core/interactiveshell.py", line 3064, in _run_cell
    result = runner(coro)
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/IPython/core/async_helpers.py", line 129, in _pseudo_sync_runner
    coro.send(None)
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/IPython/core/interactiveshell.py", line 3269, in run_cell_async
    has_raised = await self.run_ast_nodes(code_ast.body, cell_name,

```

(continues on next page)

(continued from previous page)

```

File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/IPython/core/interactiveshell.py", line 3448, in run_ast_nodes
    if await self.run_code(code, result, async_=asy):
File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↳ packages/IPython/core/interactiveshell.py", line 3508, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
File "/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_65916/3660210047.py",
↳ line 14, in <module>
    logger.info("This is an informational message")
Message: 'This is an informational message'
Arguments: ()
INFO:email_logger:This is an informational message

```

Log formatting

You can use formatters to format log messages.

```
[7]: formatter = logging.Formatter("%(asctime)s - %(name)s - %(message)s")
```

Besides `%(asctime)s`, `%(name)s` and `%(message)s` you will find other attributes in `LogRecord` attributes.

```
[8]: import logging
```

```

logger = logging.getLogger("stream_logger")
logger.setLevel(logging.INFO)

console = logging.StreamHandler()
formatter = logging.Formatter("%(asctime)s - %(name)s - %(message)s")
console.setFormatter(formatter)

logger.addHandler(console)
logger.info("This is an informational message")

This is an informational message
2023-08-03 13:40:00,195 - stream_logger - This is an informational message
INFO:stream_logger:This is an informational message

```

Note

The logging module is thread-safe. However, logging may not work in asynchronous contexts. In such cases, however, you can use the `QueueHandler`.

See also

Logging to a single file from multiple processes

Logging to multiple handlers

```
[9]: import logging

def log(path, multipleLocs=False):
    logger = logging.getLogger("Example_logger_%s" % fname)
    logger.setLevel(logging.INFO)
    fh = logging.FileHandler(path)
    formatter = logging.Formatter("%(asctime)s - %(name)s - %(message)s")
    fh.setFormatter(formatter)
    logger.addHandler(fh)

    if multipleLocs:
        console = logging.StreamHandler()
        console.setLevel(logging.INFO)
        console.setFormatter(formatter)
        logger.addHandler(console)

    logger.info("This is an informational message")
    try:
        1 / 0
    except ZeroDivisionError:
        logger.exception("You can't do that!")

    logger.critical("This is a no-brainer!")
```

Configure logging

See also

- logging configuration

... in an INI file

In the following example, the file `development.ini` is loaded in this directory:

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter

[logger_root]
level=DEBUG
handlers=stream_handler

[handler_stream_handler]
```

(continues on next page)

(continued from previous page)

```

class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s

```

```

[10]: import logging
import logging.config

from logging.config import fileConfig

logging.config.fileConfig("development.ini")
logger = logging.getLogger("example")

logger.info("Program started")
logger.info("Done!")

```

Pro:

- Ability to update the configuration on the fly by using the `logging.config.listen()` function to listen on a socket.
- Different configurations can be used in different environments, so for example, `DEBUG` can be specified as the log level in `development.ini`, while `WARN` is used in `production.ini`.

Con:

- Less control for example over custom filters or loggers configured in code.

... in a dictConfig

```

[11]: import logging
import logging.config

dictLogConfig = {
    "version": 1,
    "handlers": {
        "fileHandler": {
            "class": "logging.FileHandler",
            "formatter": "exampleFormatter",
            "filename": "dict_config.log",
        }
    },
    "loggers": {
        "exampleApp": {
            "handlers": ["fileHandler"],
            "level": "INFO",
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    },
    "formatters": {
        "exampleFormatter": {
            "format": "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
        }
    },
}

```

```
[13]: logging.config.dictConfig(dictLogConfig)
```

```
logger = logging.getLogger("exampleApp")
```

```
logger.info("Program started")
```

```
logger.info("Done!")
```

```

2021-12-12 21:22:14,326 exampleApp INFO Program started
2021-12-12 21:22:14,329 exampleApp INFO Done!

```

Pro:

- Update on the fly

Con:

- Less control than configuring a logger in code

... directly in the code

```

[12]: logger = logging.getLogger()
      handler = logging.StreamHandler()
      formatter = logging.Formatter(
          "%(asctime)s %(name)-12s %(levelname)-8s %(message)s"
      )
      handler.setFormatter(formatter)
      logger.addHandler(handler)
      logger.setLevel(logging.DEBUG)

```

Magic Commands

Befehl	Beschreibung
%logstart	Starts logging anywhere in a session %logstart [-o\ -r\ -t\ -q] [log_name [log_mode]]
	If no name is given, <code>ipython_log.py</code> is used in the current directory.
	<code>log_mode</code> is an optional parameter. The following modes can be specified:
	* <code>append</code> appends the logging information to the end of an existing file
	* <code>backup</code> renames the existing file to <code>name~</code> and writes to <code>name</code>
	* <code>global</code> appends the logging information at the end of an existing file
	* <code>over</code> overwrites an existing log file
	* <code>rotate</code> creates rotating log files: <code>name.1~</code> , <code>name.2~</code> , etc.
	Options:
	* <code>-o</code> also logs the output of IPython
	* <code>-r</code> logs raw output
	* <code>-t</code> writes a time stamp in front of each log entry
	* <code>-q</code> suppresses the logging output
%logon	Restart the logging
%logoff	Temporary termination of logging

Pro:

- Complete control over the configuration

Con:

- Changes in the configuration require a change in the source code

Logs rotate

```
[13]: import logging
import time

from logging.handlers import RotatingFileHandler

def create_rotating_log(path):
    logger = logging.getLogger("Rotating Log")
    logger.setLevel(logging.INFO)
```

(continues on next page)

(continued from previous page)

```

handler = RotatingFileHandler(path, maxBytes=20, backupCount=5)
logger.addHandler(handler)

for i in range(10):
    logger.info("This is an example log line %s" % i)
    time.sleep(1.5)

if __name__ == "__main__":
    log_file = "rotated.log"
    create_rotating_log(log_file)

```

2023-08-03 13:44:10,701	Rotating Log	INFO	This is an example log line 0
2023-08-03 13:44:10,701	Rotating Log	INFO	This is an example log line 0
2023-08-03 13:44:12,204	Rotating Log	INFO	This is an example log line 1
2023-08-03 13:44:12,204	Rotating Log	INFO	This is an example log line 1
2023-08-03 13:44:13,710	Rotating Log	INFO	This is an example log line 2
2023-08-03 13:44:13,710	Rotating Log	INFO	This is an example log line 2
2023-08-03 13:44:15,217	Rotating Log	INFO	This is an example log line 3
2023-08-03 13:44:15,217	Rotating Log	INFO	This is an example log line 3
2023-08-03 13:44:16,727	Rotating Log	INFO	This is an example log line 4
2023-08-03 13:44:16,727	Rotating Log	INFO	This is an example log line 4
2023-08-03 13:44:18,233	Rotating Log	INFO	This is an example log line 5
2023-08-03 13:44:18,233	Rotating Log	INFO	This is an example log line 5
2023-08-03 13:44:19,737	Rotating Log	INFO	This is an example log line 6
2023-08-03 13:44:19,737	Rotating Log	INFO	This is an example log line 6
2023-08-03 13:44:21,244	Rotating Log	INFO	This is an example log line 7
2023-08-03 13:44:21,244	Rotating Log	INFO	This is an example log line 7
2023-08-03 13:44:22,751	Rotating Log	INFO	This is an example log line 8
2023-08-03 13:44:22,751	Rotating Log	INFO	This is an example log line 8
2023-08-03 13:44:24,257	Rotating Log	INFO	This is an example log line 9
2023-08-03 13:44:24,257	Rotating Log	INFO	This is an example log line 9

Rotate logs time-controlled

```

[ ]: import logging
import time

from logging.handlers import TimedRotatingFileHandler

def create_timed_rotating_log(path):
    """
    logger = logging.getLogger("Rotating Log")
    logger.setLevel(logging.INFO)

    handler = TimedRotatingFileHandler(
        path, when="s", interval=5, backupCount=5
    )
    logger.addHandler(handler)

```

(continues on next page)

(continued from previous page)

```

for i in range(6):
    logger.info("This is an example!")
    time.sleep(75)

if __name__ == "__main__":
    log_file = "timed_rotation.log"
    create_timed_rotating_log(log_file)

```

```

2023-08-03 13:45:00,510 Rotating Log INFO This is an example!
2023-08-03 13:45:00,510 Rotating Log INFO This is an example!
2023-08-03 13:46:15,510 Rotating Log INFO This is an example!
2023-08-03 13:46:15,510 Rotating Log INFO This is an example!
2023-08-03 13:47:30,517 Rotating Log INFO This is an example!
2023-08-03 13:47:30,517 Rotating Log INFO This is an example!
2023-08-03 13:48:45,521 Rotating Log INFO This is an example!
2023-08-03 13:48:45,521 Rotating Log INFO This is an example!
2023-08-03 13:50:00,525 Rotating Log INFO This is an example!
2023-08-03 13:50:00,525 Rotating Log INFO This is an example!
2023-08-03 13:51:15,530 Rotating Log INFO This is an example!
2023-08-03 13:51:15,530 Rotating Log INFO This is an example!

```

Create a logging decorator

See also

- [How to Create an Exception Logging Decorator](#)

Create a logging filter

```

[ ]: import logging
import sys

class ExampleFilter(logging.Filter):
    def filter(self, record):
        if record.funcName == "foo":
            return False
        return True

logger = logging.getLogger("filter_example")
logger.addFilter(ExampleFilter())

def foo():
    """
    Ignore this function's log messages
    """
    logger.debug("Message from function foo")

```

(continues on next page)

(continued from previous page)

```
def bar():
    logger.debug("Message from bar")

if __name__ == "__main__":
    logging.basicConfig(stream=sys.stderr, level=logging.DEBUG)
    foo()
    bar()
```

7.10 Check and improve code quality and complexity

If the quality of software is neglected, this quickly leads to superfluous code, also known as cruft. This then slows down the further development of functions. This also happens to great teams who are not allowed to spend time maintaining high code quality. High code quality reduces cruft to a minimum and allows the team to add features with less effort, time and cost. Although there are some indicators that can be used to measure internal quality, these can only provide an initial indication of further productivity. However, a recent [study](#) indicates that low quality code took more than twice as long to fix as high quality code, and that low quality code had a 15 times higher defect density.

In the following, I will show you some *Code-Smells and design principles* and then some tools with which you can perform automated static analyses and reformat the code. You can integrate some of these tools into your editor as well as via the *pre-commit framework*. Finally, I'll introduce you to *Rope*, a tool that supports you with refactorings.

See also:

- [PyCQA Meta Documentation](#)
- github.com/PyCQA

7.10.1 Code-Smells and design principles

Code smells are coding patterns that indicate that something is wrong with the design of a programme. For example, the overuse of isinstance checks against concrete classes is a code smell, as it makes the programme more difficult to extend to deal with new types in the future.

Recognising code smells

One way to better recognise code smells is to describe the characteristics of code. Make a note of the things you recognise; add any patterns you see, like or don't understand. The following questions may prompt you to think further:

- Are there methods that have the same form?
- Are there methods that have an argument with the same name?
- Do arguments with the same name always mean the same thing?
- If you want to add a private method to a class, where would it go?
- If you were to split the class into two parts, where would the dividing line be?
- Do the tests in the conditions have anything in common?
- How many branches do the conditions have?

- Do the methods contain any code other than the condition?
- Does each method depend more on the argument passed or on the class as a whole?

SOLID principles

SOLID is an acronym for:

S – *Single responsibility principle*

The methods of a class should be orientated towards a single purpose.

O – *Open-closed principle*

Objects should be open for extensions but closed for changes.

L – *Liskov's principle of substitution*

Subclasses should be substitutable by their superclasses.

I – *Interface segregation principle*

Objects should not depend on methods that they do not use.

D – *Dependency inversion principle*

Abstractions should not depend on details.

Open-closed principle

The decision as to whether refactoring should be carried out at all should depend on whether your code is already *open* to new requirements. Open here means that your code should be open for extensions without having to change existing code. Refactorings should not be mixed with the addition of new functions. Instead, these two processes should be kept separate. When faced with a new requirement, first reorganise the existing code so that it is open to the new function and only add the new code once this has been completed.

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

– Martin Fowler: Refactoring

Note: Safe refactoring relies on [tests](#). If you really refactor the code without changing the behaviour, the existing tests should continue to succeed at every step. The tests are a safety net that justifies confidence in the new arrangement of the code. If they fail,

- you have inadvertently broken the code,
 - or the existing tests are flawed.
-

Single responsibility principle

The [single responsibility principle](#) states that each class should only fulfil one task:

There should never be more than one reason for a class to change.

– Robert C. Martin: SRP: The Single Responsibility Principle

Liskov's principle of substitution

The [Liskov substitution principle](#) states that subclasses must be substitutable by their superclasses. The Liskov substitution principle also applies to [Duck typing](#): every object that claims to be a duck must fully implement the duck's API. Duck types should be interchangeable. Applying logic across different data types of objects is called [polymorphism](#).

Interface segregation principle

the [interface segregation principle](#) applies the *Single responsibility principle* to interfaces in order to isolate a specific behaviour. If a change to a part of your code is required, extracting an object that plays a role opens up the possibility of supporting the new behaviour without having to change the existing code. This is preferable to coded concretisations.

In this context, [Demeter's law](#) is also interesting, which states that objects should only communicate with objects in their immediate environment. This effectively restricts the list of other objects to which an object can send a message and reduces the coupling between objects: an object can only talk to its neighbours, but not to the neighbours of its neighbours; objects can only send messages to those directly involved.

Dependency inversion principle

The [Dependency inversion principle](#) can be defined as

Abstractions should not depend upon details. Details should depend upon abstractions.

– Robert C. Martin: The Dependency Inversion Principle

Typical code smells in Python

Functions that should be objects

In addition to object-oriented programming, Python also supports procedural programming using functions and inheritable classes. Both paradigms should, however, be applied to the appropriate problems.

Typical symptoms of functional code that should be converted to classes are

- similar arguments across functions
- high number of distinct Halstead operands
- mix of mutable and immutable functions

For example, three functions with ambiguous usage can be reorganised so, that `load_image()` is replaced by `__init__()`, `crop()` becomes a class method, and `get_thumbnail()` a property:

```
class Image(object):
    thumbnail_resolution = 128

    def __init__(self, path):
        ...

    def crop(self, width, height):
        ...

    @property
    def thumbnail(self):
```

(continues on next page)

(continued from previous page)

```
...  
return thumb
```

Objects that should be functions

Sometimes, however, object-oriented code should also be better broken down into functions, for example if a class contains only one other method apart from `__init__()` or only static methods.

Note: You do not have to search for such classes manually, but there is a pylint rule for it:

```
$ pipenv run pylint --disable=all --enable=R0903 requests  
***** Module requests.auth  
requests/auth.py:72:0: R0903: Too few public methods (1/2) (too-few-public-methods)  
requests/auth.py:100:0: R0903: Too few public methods (1/2) (too-few-public-methods)  
***** Module requests.models  
requests/models.py:60:0: R0903: Too few public methods (1/2) (too-few-public-methods)  
  
-----  
Your code has been rated at 9.99/10
```

This shows us that two classes with only one public method have been defined in `auth.py`, in lines 72ff. and 100ff. Also in `models.py` there is a class with only one public method from line 60.

Nested code

«Flat is better than nested.»

– Tim Peters, [Zen of Python](#)

Nested code makes it difficult to read and understand. You need to understand and remember the conditions as you go through the nestings. Objectively, the cyclomatic complexity increases as the number of code branches increases.

You can reduce nested methods with multiple nested `if` statements by replacing levels with methods that return `False` if necessary. Then you can use `.count()` to check if the number of errors is `> 0`.

Another possibility is to use list comprehensions. This way the code

```
results = []  
for item in iterable:  
    if item == match:  
        results.append(item)
```

can be replaced by

```
results = [item for item in iterable if item == match]
```

Note: The [itertools](#) of the Python standard library are often also good for reducing the nesting depth by creating functions to create iterators from data structures.

Note: You can also filter with `itertools`, for example with `filterfalse`:

```
>>> from itertools import filterfalse
>>> from math import isnan
>>> from statistics import median
>>> data = [20.7, float('NaN'), 19.2, 18.3, float('NaN'), 14.4]
>>> sorted(data)
[20.7, nan, 14.4, 18.3, 19.2, nan]
>>> median(data)
16.35
>>> sum(map(isnan, data))
2
>>> clean = list(filterfalse(isnan, data))
>>> clean
[20.7, 19.2, 18.3, 14.4]
>>> sorted(clean)
[14.4, 18.3, 19.2, 20.7]
>>> median(clean)
18.75
```

Query tools for complex dicts

`JMESPath`, `glom`, `asq` and `flupy` can significantly simplify the query of dicts in Python.

Reduce code with dataclasses and attrs

dataclasses

are intended to simplify the definition of classes that are mainly created to store values and can then be accessed via attribute search. Some examples are `collections.namedtuple()`, `typing.NamedTuple`, recipes for `records` and `nested dicts`. Data classes save you from having to write and manage these methods.

See also:

- [PEP 557](#) – Data Classes

attrs

is a Python package that has been around much longer than `dataclasses`, is more comprehensive and can also be used with older versions of Python.

See also:

- [Effective Python](#) by Brett Slatkin
- [When Python Practices Go Wrong](#) by Brandon Rhodes

7.10.2 Checker

flake8

is a wrapper around [PyFlakes](#), [pycodestyle](#) and [McCabe](#). However, automatic formatting, for example with [Black](#), is even more convenient.

Mypy

is a static type checker.

Pytype

is a static analysis tool that derives types from your Python code without the need for type annotations.

Wily

is a command line tool for checking the complexity of Python code in tests and applications.

Pystra

analyses the structural reliability of Python code and summarises it in a report.

Pysa

performs [taint](#) analysis to identify potential security problems. Pysa traces data streams from their origin to their endpoint and identifies vulnerable code.

check-manifest

is a tool with which you can quickly check whether the file [MANIFEST.in](#) for Python packages is complete.

flake8

flake8 is a wrapper around [PyFlakes](#), [pycodestyle](#) and [McCabe](#). However, automatic formatting, for example with [Black](#), is even more convenient.

Installation

```
$ spack env activate python-311
$ spack install py-flake8
```

Check

```
$ flake8 PATH/TO/YOUR/CODE
```

Configuration

flake8 can be configured for **tox** in the **tox.ini** file of a package, for example:

```
[tox]
envlist = py38, py311, flake8, docs

[testenv:flake8]
basepython = python
deps =
    flake8
    flake8-isort
```

(continues on next page)

(continued from previous page)

```
commands =  
    flake8 src tests setup.py conftest.py docs/conf.py
```

See also:

- [Configuring flake8](#)
- [flake8 error/violation codes](#)
- [pycodestyle error codes](#)

Mypy

With [Mypy](#) you can do a static type check.

See also:

- [Home](#)
- [GitHub](#)
- [Docs](#)
- [PyPI](#)
- [Using Mypy in production at Spring](#)

Installation

Mypy requires Python3.5. Then it can be installed, for example with:

```
$ pipenv install mypy
```

Check

Then you can check it, for example with:

```
$ pipenv run mypy myprogram.py
```

Note: Although Mypy needs to be installed with Python3, it can also parse Python2 code, for example with:

```
$ pipenv run mypy --py2 myprogram.py
```

Pytype

Pytype is a static analysis tool that derives types from your Python code without the need for type annotations. However, it can also enforce [type annotations](#) that are in the code. Although annotations are optional for Pytype, they are checked and applied if they are present. The type annotations generated by Pytype are stored in standalone `.pyi` files, which can be merged back into Python using [merge-pyi](#). Finally, it flags common errors such as misspelled attribute names or function calls and much more, even across file boundaries.

See also:

- [Home](#)
- [GitHub](#)
- [PyPI](#)
- [User guide](#)
- [FAQ](#)

Requirements

- All common Linux distributions are supported
- macOS 10.7 and Xcode 8
- Windows with [WSL](#). In addition, the following libraries must be installed:

```
$ sudo apt install build-essential python3-dev libpython3-dev
```

Installation

Pytype can be easily installed with

```
$ pipenv install pytype
```

The installation can then be checked with

```
$ pipenv run pytype file_or_directory
```

Configuration

For a Python package, you can set up Pytype by creating a `pytype.cfg` file with

```
$ pipenv run pytype --generate-config pytype.cfg
```

This then starts with for example

```
# NOTE: All relative paths are relative to the location of this file.

[pytype]

# Space-separated list of files or directories to exclude.
exclude =
```

(continues on next page)

(continued from previous page)

```

**/*_test.py
**/test_*.py

# Space-separated list of files or directories to process.
inputs =
.

```

Now you can customise the configuration file according to your requirements.

Additional scripts

annotate-ast

in-progress type annotator for ASTs

merge-pyi

Merge type information from a .pyi file into a Python file

pytd-tool

parser for .pyi files

pytype-single

debugging tool for pytype developers that analyses a single python file assuming that .pyi files have already been generated for all dependencies

pyxref

cross-references generator

Wily

The *Zen of Python*¹ emphasises complexity reduction in many ways:

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Wily is a command line tool for checking the complexity of Python code in tests and applications. For this purpose, Wily uses the following metrics:

Cyclomatic complexity

measures the complexity of code by the number of linearly independent paths in the control flow graph.

The Software Engineering Institute at Carnegie Mellon University distinguishes the following four levels of risk²:

Cyclomatic complexity	Risk assessment
1–10	Simple programme without much risk
11–20	moderate risk
21–50	complex, high-risk programme
> 50	untestable programme with very high risk

¹ **PEP 20** – The Zen of Python

² C4 Software Technology Reference Guide, S. 147

Halstead complexity measures

Statically analysing procedure that calculates the difficulty of the programme, the effort and the implementation time from the number of operators and operands.

Maintainability Index

is based on the cyclomatic complexity, the Halstead complexity measures and the number of lines of code³:

Index	Maintainability
0–25	unmaintainable
25–50	worrying
50–75	in need of improvement
75–100	Superhero code

See also:

- [Docs](#)
- [GitHub](#)
- [PyPI](#)
- [wily-pycharm](#)

Installation

Wily can be easily installed with

```
$ pipenv install wily
```

You can then check the installation with

```
$ pipenv run wily --help
Usage: wily [OPTIONS] COMMAND [ARGS]...
Version: 1.19.0
  Inspect and search through the complexity of your source code. To get
  started, run setup:
  $ wily setup ...
```

Configuration

A `wily.cfg` file can be created in the project directory with the list of available operators:

```
[wily]
# list of operators, choose from cyclomatic, maintainability, mccabe and raw
operators = cyclomatic,raw
# archiver to use, defaults to git
archiver = git
# path to analyse, defaults to .
path = /path/to/target
# max revisions to archive, defaults to 50
max_revisions = 20
```

³ Using Metrics to Evaluate Software System Maintainability

Python code in `.ipynb` files is also usually recognised automatically. However, you may be able to disable this for a Jupyter notebook with

```
ipynb_support = false
```

or for individual cells with

```
ipynb_cells = false
```

Use

... as a command line tool

1. Building a cache with the statistics of the project

Note: Wily assumes that your project folder is a *Git* repository. However, Wily does not create a cache if the working directory is dirty.

```
$ pipenv run wily build
```

2. Show metric

```
$ pipenv run wily report
```

This outputs both the metric and the delta to the previous revision.

3. Show ranking

```
$ pipenv run wily rank
```

This shows the ranking of all files in a directory or a single file based on the specified metric, if present in `.wily/`.

4. Show graph

```
$ pipenv run wily graph
```

This displays a graph in the default browser.

5. Show build directory information

```
$ pipenv run wily index
```

6. List the metrics available in the Wily operators

```
$ pipenv run wily list-metrics
```

... as pre-commit hook

You can also use Wily as a *pre-commit framework*. To do this, you would have to add the following to the `pre-commit-config.yaml` configuration file, for example:

```
repos:
-   repo: local
    hooks:
    -   id: wily
        name: wily
        entry: wily diff
        verbose: true
        language: python
        additional_dependencies: [wily]
```

... in a CI/CD pipeline

Usually Wily compares the complexity with the previous revision. However, you can also specify other references, for example `HEAD^1` with

```
$ pipenv run wily build src/
$ pipenv run wily diff src/ -r HEAD^1
```

Pystra

PYSTR (Python Structural Reliability Analysis) analyses the structural reliability of Python code and summarises it in a report.

See also:

- [Docs](#)
- [GitHub](#)

Installation

```
$ pipenv install pystra
```

Reliability analysis

A FORM (first-order reliability method) analysis can lead to the following result, for example:

```
=====

RESULTS FROM RUNNING FORM RELIABILITY ANALYSIS

Number of iterations:      17
Reliability index beta:    1.75397614074
```

(continues on next page)

(continued from previous page)

```
Failure probability:      0.039717297753
Number of calls to the limit-state function: 164
```

Pysa

The Python Static Analyzer Pysa performs [taint](#) analysis to identify potential security problems. Pysa traces data streams from their origin to their endpoint and identifies vulnerable code.

See also:

- [What Is Taint Analysis and Why Should I Care?](#)
- [How Pysa works](#)
- [Running Pysa](#)
- [Pysa Tutorial](#)

Configuration

Pysa uses two file types for configuration:

- a `taint.config` file in JSON format, in which sources, sinks, features and rules are defined.

```
{
  "comment": "UserControlled, Test, Demo sources are predefined. Same for Demo, ↵
↵Test and RemoteCodeExecution sinks",
  "sources": [],
  "sinks": [],
  "features": [],
  "rules": []
}
```

- files with the extension `.pysa` in a directory configured with `taint_models_path` in your `.pyre_configuration` file.

You can find practical examples in the [Pyre repository](#).

Use

Pyre can be called, for example with

```
$ $ pipenv run pyre analyze --save-results-to ./
```

The `--save-results-to` option stores detailed results in `./taint-output.json`.

Pysa postprozessor

Installation

```
$ pipenv install fb-sapp
```

Use

1. Parsing the JSON file, for example with

```
$ pipenv run sapp --database-name sapp.db analyze ./taint-output.json
```

The results are stored in the local SQLite file `sapp.db`.

2. Exploring the problems with

```
$ pipenv run sapp --database-name sapp.db explore
```

This starts an IPython interface connected to the SQLite database:

issues

lists all issues

issue 1

selects the first issue

trace

shows the data flow from source to sink

n

jumps to the next call

list

shows the source code of the call

jump 1

jumps to the first call and shows the source code

Further commands can be found in the [SAPP Command-Line Interface](#).

check-manifest

`check-manifest` is a tool with which you can quickly check whether the file `Manifest.in` for Python packages is complete.

Installation

```
$ pipenv install check-manifest
```

Check

```
$ cd /path/to/MANIFEST.in
$ pipenv run check-manifest
```

... or for an automatic update

```
$ pipenv run check-manifest -uv
listing source files under version control: 6 files and directories
building an sdist: check-manifest-0.7.tar.gz: 4 files and directories
lists of files in version control and sdist do not match!
missing from sdist:
  tests.py
  tox.ini
suggested MANIFEST.in rules:
  include *.py
  include tox.ini
updating MANIFEST.in

$ cat MANIFEST.in
include *.rst

# added by check_manifest.py
include *.py
include tox.ini
```

Configuration

You can configure `check-manifest` so that certain file patterns are ignored by creating a section `[tool.check-manifest]` in your `pyproject.toml` file or a section `[check-manifest]` in your `setup.cfg` or `tox.ini` file, for example:

```
[tool.check-manifest]
ignore = [".travis.yml"]

# setup.cfg or tox.ini
[check-manifest]
ignore =
    .travis.yml
```

`check-manifest` knows the following options:

ignore

A list of filename patterns that are ignored by `check-manifest`. Use this option if you want to keep files in your version control system that shouldn't be in your source distributions. The standard list is:

```
PKG-INFO
* .egg-info
* .egg-info / *
setup.cfg
.hgtags
.hgsigs
.hgignore
.gitignore
.bzrignore
.gitattributes
.github / *
.travis.yml
Jenkinsfile
* .mo
```

ignore-default-rules

If true, your ignore entries replace the standard list instead of completing it.

ignore-bad-ideas

A list of filename patterns that will be ignored by checking the generated files. This allows you to keep generated files in your version control system, even if this is usually a bad idea.

Integration with version control

With *pre-commit framework*, *check-manifest* can be part of your Git workflow. To do this, add the following to your *.pre-commit-config.yaml* file:

```
repos:
-   repo: https://github.com/mgedmin/check-manifest
    rev: "0.39"
    hooks:
    -   id: check-manifest
```

7.10.3 Formatter

Black

formats your code in a nice and deterministic format.

isort

formats your `import` statements in separate and sorted blocks.

prettier

offers automatic formatters for other file types.

Black

[Black](#) formats your code in a nice and deterministic format.

See also:

Was lesbaren Code auszeichnet, ist gut beschrieben im Trey Hunners Blog-Post [Craft Your Python Like Poetry](#).

Installation

```
$ pipenv install black
```

Check

Then you can check the installation with

```
$ pipenv run black /PATH/TO/YOUR/SOURCE/FILE
```

Integration

With [jupyter-black](#) you can already use Black in your Jupyter notebooks.

See also:

Integration into other editors such as PyCharm, Wing IDE or Vim is also possible, see [Editor integration](#)

Configuration

In contrast to Black's standard 88-character formatting, however, I prefer a line length of 79 characters.

For this you can enter the following in `pyproject.toml`:

```
[tool.black]
line-length = 79
```

See also:

You can get more information about the configuration of Black in the Toml file in [pyproject.toml](#).

isort

[isort](#) formats your `import` statements in separate and sorted blocks.

Installation

```
$ pipenv install isort
```

Configuration

isort can be configured for example in the `pyproject.toml` file:

```
[tool.isort]
atomic=true
force_grid_wrap=0
include_trailing_comma=true
lines_after_imports=2
lines_between_types=1
multi_line_output=3
not_skip="__init__.py"
use_parentheses=true

known_first_party=["MY_FIRST_MODULE", "MY_SECOND_MODULE"]
known_third_party=["mpi4py", "numpy", "requests"]
```

In order to recognise third-party packages for your project imports, you can install your project together with `isort`.

Note: With `isort 5` you can use profiles. This simplifies the configuration of `isort` in order to continue to play with *Black* in the future:

```
isort --profile black .
```

prettier

`prettier` offers automatic formatters for other file types, including `TypeScript`, `JSON`, `Vue`, `YAML`, `TOML` and `XML`.

Installation

```
$ npm install prettier --save-dev --save-exact
```

Configuration

```
$ npx prettier --write path/to/my/file.js
```


Pre-commit hook for prettier

Installation

```
$ npm install pretty-quick husky --save-dev
```

Configuration

In the `package.json` file you can configure the pre-commit hook as follows:

```
{ "husky": { "hooks": { "pre-commit": "pretty-quick --staged" } } }
```

See also:

- [Prettier docs](#)

7.10.4 Refactoring

Rope

is a Python refactoring library.

Rope

Rope is a Python refactoring library.

Installation

Rope can be easily installed with

```
$ pipenv install rope
```

Use

Now we first import the `Project` type and instantiate it with the path to the project:

```
[1]: from rope.base.project import Project

proj = Project("requests")
```

This creates a project folder named `.ropeproject` in our project.

```
[2]: [f.name for f in proj.get_files()]
```

```
[2]: ['hooks.py',
      'utils.py',
      '_internal_utils.py',
      'status_codes.py',
      '__version__.py',
```

(continues on next page)

(continued from previous page)

```
'sessions.py',
'api.py',
'cookies.py',
'adapters.py',
'certs.py',
'exceptions.py',
'api_v1.py',
'auth.py',
'help.py',
'structures.py',
'compat.py',
'packages.py',
'__init__.py',
'models.py']
```

The `proj` variable can execute a number of commands such as `get_files` and `get_file`. In the following example we use this to assign the variable `api` to the file `api.py`.

```
[3]: !cp requests/api.py requests/api_v1.py
```

```
[4]: api = proj.get_file("api.py")
```

```
[5]: from rope.refactor.rename import Rename

change = Rename(proj, api).get_changes("api.py")

proj.do(change)
```

```
[6]: !cd requests && git status
```

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   __init__.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .ropeproject/
        api_v1.py

Changes not staged for commit (use "git add" and/or "git commit -a")
```

```
[7]: !cd requests && git diff __init__.py
```

```
diff --git a/__init__.py b/__init__.py
index f8f9429..502e33a 100644
--- a/__init__.py
+++ b/__init__.py
@@ -118,7 +118,7 @@ from __version__ import __copyright__, __cake__
 from . import utils
 from . import packages
```

(continues on next page)

(continued from previous page)

```

from .models import Request, Response, PreparedRequest
-from .api import request, get, head, post, patch, put, delete, options
+from .api_v1 import request, get, head, post, patch, put, delete, options
from .sessions import session, Session
from .status_codes import codes
from .exceptions import (

```

With `proj.do(change)`, the file `requests/__init__.py` has been changed to import from `new_api` instead of `api`.

Rope can be used not only for renaming files, but also in various other cases; see also [Rope Refactorings](#).

See also:

- [Martin Fowler: Refactoring](#)

7.11 Security

In the previous chapters we have already given some hints that enable a safer operation. Here we want to summarise and expand the individual elements. In doing so, we will be guided by the [OpenSSF Scorecard](#). Alternatively, you can also follow [ISO/IEC 5230/OpenChain](#).

7.11.1 Check vulnerabilities

Risk: High

This check determines whether the project has open, unfixed vulnerabilities in its own code base or in its dependencies. An open vulnerability can be easily exploited and should be closed as soon as possible.

For such a check, you can use for example `pipenv check`, which uses the Python library `safety`. Alternatively, you can use `osv` or `pip-audit`, which uses the [Open Source Vulnerability Database](#).

If a vulnerability is found in a dependency, you should update to a non-vulnerable version; if no update is available, you should consider removing the dependency.

If you believe that the vulnerability does not affect your project, an `osv-scanner.toml` file can be created for `osv`, including the ID to ignore and a reason, for example:

```

[[IgnoredVulns]]
id = "GO-2022-1059"
# ignoreUntil = 2022-11-09 # Optional exception expiry date
reason = "No external http servers are written in Go lang."

```

7.11.2 Maintenance

Are the dependencies updated automatically?

Risk: High

Outdated dependencies make a project vulnerable to attacks on known vulnerabilities. Therefore, the process of updating dependencies should be automated by checking for outdated or insecure requirements and updating them if necessary. You can use [dependabot](#) or [PyUp](#) for this purpose.

You can also update your `Pipenv` environments automatically with `pipenv update`.

Are the dependencies still maintained?

Risk: High

This indicates possible unpatched security vulnerabilities. Therefore, it should be checked regularly whether a project has been archived. Conversely, the OSSF scorecard assumes that with at least one commit a week for 90 days, the project is very actively maintained. However, a lack of active maintenance is not necessarily always a problem: smaller utilities in particular usually do not need to be maintained, or only very rarely. So a lack of active maintenance only tells you that you should investigate the situation more closely.

You can also display the activities of a project with badges, for example:

Is there a safety concept for the project?

Risk: Medium

Ideally, a `SECURITY.md` or similar file should have been published with the project. This file should contain information

- how a security vulnerability can be reported without it becoming publicly visible,
- on the procedure and schedule for disclosing the vulnerability,
- to links, for example URLs and emails, where support can be requested.

See also:

- [Guide to implementing a coordinated vulnerability disclosure process for open source projects](#)
- [Adding a security policy to your repository](#)
- [Runbook](#)

Does the project contain a usable licence?

Risk: Low

A [license](#) indicates how the source code may or may not be used. The absence of a licence complicates any kind of security review or audit and poses a legal risk for potential use.

OSSF-Scorecard uses the [GitHub License API](#) for projects hosted on GitHub, otherwise it uses its own heuristics to detect a published license file. Files in a `LICENSES` directory should be named with their [SPDX](#) licence identifier followed by an appropriate file extension as described in the [REUSE](#) specification.

Are the best practices of the OPENSSF (Open Source Security Foundation) being followed?

Risk: Low

The [Open Source Security Foundation \(OpenSSF\) Best Practices Program](#) includes a set of security-oriented best practices for open source software development:

- the vulnerability reporting procedure is published on the project page
- a working build system automatically rebuilds the software from source code
- a general policy that tests are added to an automated test suite when important new features are added
- various cryptography criteria are met, if applicable
- at least one static code analysis tool applied to each planned major production release

You can also get a corresponding badge with the [OpenSSF Best Practices Badge Programm](#).

7.11.3 Continuous testing

Are CI tests carried out in the project?

Risk: Low

Before code is merged into pull or merge requests, tests should be performed to help detect errors early and reduce the number of vulnerabilities in a project.

Does the project use fuzzing tools?

risk: Medium

Fuzzing or fuzz testing passes unexpected or random data to your programme to detect bugs. Regular fuzzing is important to detect vulnerabilities that can be exploited by others, especially since fuzzing can also be used in an attack to find the same vulnerabilities.

- Does your project use [fuzzing](#)?
- Is the name of the repository included in the [OSS fuzz](#) project list?
- Is [ClusterFuzzLite](#) used in the repository?
- Are custom language-specific fuzzing features present in the repository, for example with [atheris](#) or [OneFuzz](#)?

Does your project use static code analysis tools?

Risk: Medium

[Static code analysis](#) tests the source code before the application is executed. This can prevent known bug classes from being accidentally introduced into the codebase.

To check for vulnerabilities, you can use [bandit](#), which you can also integrate into your `.pre-commit-hooks.yaml`:

```
repos:
- repo: https://github.com/PyCQA/bandit
  rev: '1.7.5'
  hooks:
  - id: bandit
```

You can also use [Pysa](#) for [taint](#) analyses.

For GitHub repositories you can also use [CodeQL](#); see [codeql-action](#).

7.11.4 Risk assessment of the source code

Is the project free of checked-in binaries?

Risk: High

Generated executables in the source code repository (for example Java `.class` files, Python `.pyc` files) increase risk because they are difficult to verify, so they may be out of date or maliciously tampered with. These problems can be countered with verified, reproducible builds, but their executables should not end up back in the source code repository.

Is the development process vulnerable to the introduction of malicious code?

Risk: High

With *protected Git branches*, rules can be defined for the adoption of changes in standard and release branches, for example automated static code analyses with *flake8*, *Pysa*, *Wily* and *code reviews* via *merge requests*.

See also:

- Reproducible Builds
- Python 3.12.0 from a supply chain security perspective
- Defending against the PyTorch supply chain attack PoC

Are code reviews performed?

Risk: High

Code reviews can detect unintentional vulnerabilities or possible introduction of malicious code. Possible attacks can be detected in which the account of a team member has been infiltrated.

Does the project involve people from several organisations?

Risk: Low

This is taken as an indication of a lower number of trustworthy code reviewers. For this purpose, you can search for different entries in the `* Company*` field in the profiles. At least three different companies in the last 30 commits are desirable, whereby each of these team members should have made at least five commits.

7.11.5 Risk assessment of the builds

Are dependencies declared and fixed in the project?

Risk: Medium

In your project, dependencies used during the build and release process should be pinned. A pinned dependency should be explicitly set to a specific hash and not just to a mutable version or version range.

Spack writes these hashes for the respective environment in *spack.lock*, *Pipenv* in *Pipfile.lock*. These files should therefore also be checked in with the source code.

This can reduce the following security risks:

- Testing and deployment are done with the same software, which reduces deployment risks, simplifies debugging and enables reproducibility.

- Compromised dependencies do not undermine the security of the project.
- Substitution attacks, i.e. (id est) attacks that aim to confuse dependencies, can thus be countered.

However, fixing dependencies should not prevent software updates. You can reduce this risk by

- automated tools that notify you when dependencies in your project are out of date
- update applications that lock dependencies quickly.

CREATE WEB APPLICATIONS

I will introduce you to three different types of web applications:

- [Dashboards](#) generated from Jupyter notebooks
- Web applications that go beyond notebooks, such as integrating bokeh plots, as demonstrated in [Bokeh-Plots in Flask einbinden](#)
- Finally, the provision of your data via a [RESTful API](#), for example with the *FastAPI framework*.

CHAPTER
NINE

INDEX

Symbols

```
$ git log MAIN..FEATURE, 386
$ git log -G"BA*", 385
$ git log -L :FUNCNAME_REGEX:PATH/TO/FILE, 386
$ git log -L LINE_START_INT|LINE_START_REGEX,LINE_END_INT|LINE_END_REGEX:PATH/TO/FILE, 386
$ git log -S"FOO" [-i], 385
$ git log -- PATH, 385
$ git log --author="VEIT", 385
$ git log --follow PATH/TO/FILE, 386
$ git log --grep="TERM" [-i], 385
$ git log --oneline --decorate --graph --all|FEATURE, 386
$ git log --reverse, 386
$ git log --stat --patch|-p, 386
$ git log [--after="YYYY-MM-DD"] [--before="YYYY-MM-DD"], 385
$ git log [-n COUNT], 385
$ git reflog, 387
```

A

ACID, 273

B

BASE, 273
Branch, 468

C

Cache, 468
CAP theorem, 274
Cassandra, 274
Clone, 468
Column Family, 274
Commit, 468
Consistency, 274
Consistent hash function, 274
CouchDB, 275

E

Eventual Consistency, 275

F

Fork, 469

G

Git, 469
Graph model, 275
Graph partitioning, 275
Graph traversal, 275

H

HBase, 275
HEAD, 469
Hypertable, 275

I

Index, 469

K

Key/value pair, 275

L

Locking, 276

M

MapReduce, 276
Merge request, 469
MongoDB, 276
MVCC - Multiversion Concurrency Control, 276

O

Optimistic concurrency, 276
origin, 469

P

Paxos, 276
Pessimistic locking, 276
PGM, 276
Property graph model, 276
Python Enhancement Proposals
PEP 0282, 535

- PEP 20, [555](#)
- PEP 249, [217](#)
- PEP 257, [439](#)
- PEP 307, [185](#)
- PEP 3154, [185](#)
- PEP 503, [522](#)
- PEP 508, [506](#)
- PEP 557, [551](#)
- PEP 621, [522](#)
- PEP 639, [522](#)
- PEP 643, [522](#)
- PEP 658, [522](#)
- PEP 659, [344](#)

R

Redis, [276](#)

Remote repository, [469](#)

RFC

- RFC 4122, [267](#)
- RFC 4180, [162](#)
- RFC 4506, [188](#)
- RFC 7158#section-9, [172](#)
- RFC 7159, [172](#)
- RFC 8259, [172](#)

Riak, [276](#)

S

Semantic integrity, [276](#)

T

TBD, [469](#)

Trunk-Based Development, [469](#)

Two-phase locking (2PL), [276](#)

V

Vector clock, [277](#)

W

Working Tree, [469](#)

X

XPATH, [277](#)

XQuery, [277](#)

XSLT, [277](#)