Python basics

Release 24.1.0

Veit Schiele

Apr 11, 2024

CONTENTS

Welcome to Python Basics! I have written this book to provide an easy and practical introduction to Python. The book is not intended to be a comprehensive reference guide to Python, but rather the goal is to give you a basic familiarity with Python and enable you to quickly write your own programs.

Note: If you have suggestions for improvements, I would be happy to receive them.

CHAPTER

ONE

INTRODUCTION

1.1 About Python

You may be asking yourself why you should learn Python. There are many programming languages from C and C++ to Java, Lua and Go.

Fig. 1: TIOBE Index für Oktober 2022

Python has become very widely used and one of the reasons might be that it runs on many different platforms, from IoT devices to common operating systems to supercomputers. It can be used well for developing small applications and fast prototypes. In the process, there are countless software libraries to make your work easier.

Python is a modern programming language developed by Guido van Rossum in the 1990s.

See also:

- The Origins of Python by Lambert Meertens

Some strengths of Python are

ease of use
: Some of the reasons for this are that types are associated with objects, not variables; a variable can be assigned values of any type and a list can contain objects of different types. Also, the syntax rules are very simple and you can quickly learn to write useful code.

Expressive power
: Often you can achieve much more in a few lines of code than in other languages. As a result, you can complete your projects more quickly, and debugging and maintenance are also much easier.

Readability
: The easy readability of Python code simplifies debugging and maintenance. One of the ways Python achieves this is by requiring indentation.

Completeness
: With the installation of Python, everything essential needed for programming with Python is already available, emails, websites, databases, without the need to install additional libraries.

Platform independence
: Python runs on many platforms: Windows, Mac, Linux ETC /ET CETERA). There are even variants that run on Java (Jython) and .NET (IronPython).

Open Source
: You can download Python and use it freely for developing commercial or private applications. Python is used and promoted by many established companies, including Google, Meta and Bloomberg. And if you want to give something back, you are also welcome to do so : Python Software Foundation Sponsorship

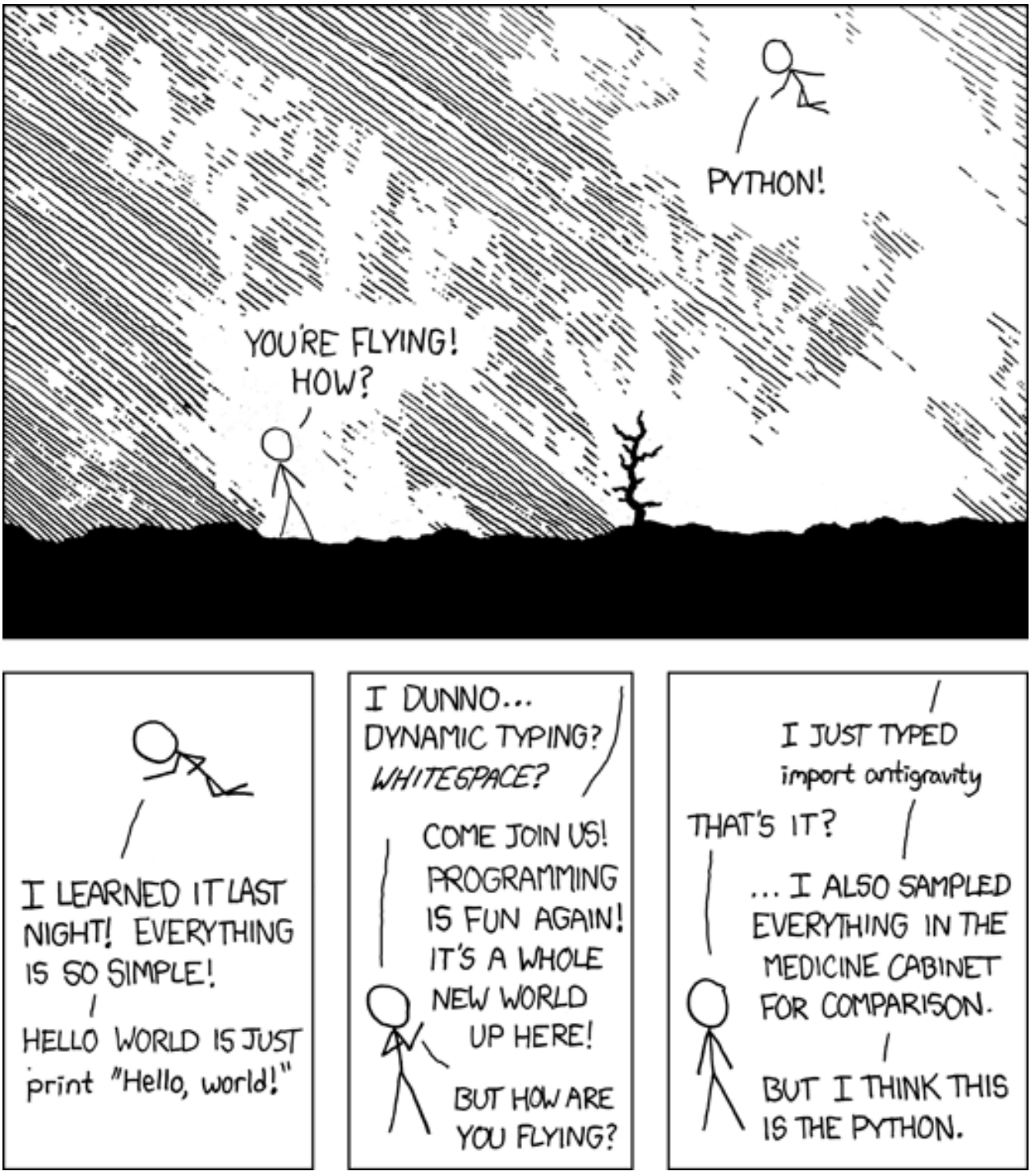


Fig. 2: XKCD: Python

Python has some advantages, but no language is the best solution in all areas. For example, Python performs less well in the following areas:

Speed
: Python is not a fully compiled language and code is first compiled into bytecode before being executed by the Python interpreter. While there are some tasks, such as string parsing with regular expressions, for which Python provides efficient implementations, and which are as fast as a C program, Python programs will still be slower than C programs in most cases. However, this rarely plays a decisive role, since there are already many Python modules that use C internally.

  See also:

  - Performance

Diverse libraries
: Python already has a lot of libraries, but in some cases you will only find suitable libraries in other languages. For most problems that need to be solved programmatically, however, Python's library support is excellent.

Variable types
: Unlike in many other languages, variables are not containers, but rather labels that refer to various objects: Integers, strings, class instances and more. Some find it a disadvantage that Python does not simply perform type validation here, but the number of type errors is usually manageable and the flexibility of dynamic typing usually outweighs the problems.

Support for mobile devices
: Even though mobile devices have proliferated in recent years, Python does not have a strong presence in this area. While there are a few options to deploy and run Python on mobile devices, this is not always easy.

Support for concurrent computation
: Processors with multiple cores are now widespread and lead to significant performance gains in many areas. However, the standard implementation of Python is not designed to use multiple cores.

  See also:

  - Introduction to multithreading, multiprocessing and async

CHAPTER

TWO

INSTALLATION

The installation of Python is simple. The first step is to download the latest version from www.python.org/downloads. The tutorial is based on Python 3.10, but if you have Python 3.7 or 3.8 installed, that is no problem either.

Most Linux distributions have Python already installed. If a precompiled version of Python exists in your Linux distribution, I recommend you to use it.

You need a Python version that matches your macOS and processor. Once you have determined the correct version, you can download the image file, mount it with a double click and then start the installation programme contained in it. Python will then be in the `Applications` folder.

If you use Homebrew, you can also install Python in the terminal with:

```
$ brew install python3
```

Python can be installed for most Windows versions after Windows 7 with the Python installer in three steps:

1. Download the latest Python Releases for Windows installer, for example Windows installer (64-bit).
2. Start the installation programme. If you have the necessary permissions, install Python with the option *Install launcher for all users*. This should install Python in `C:\Program Files\Python310-64`. Also, *Add Python 3.10 to PATH* should be activated so that this path to the Python installation is also entered in the list of `PATH` environment variables.
3. Finally, you can now check the installation by entering the following in the command prompt:

   ```
   C:\> python -V
   Python 3.10.6
   ```

Note: If Python is already installed on your system, you can easily install your own Python. A new version does not replace the old one but is installed in a different location.

CHAPTER

THREE

EDITORS

3.1 Interactive Shell

With the interactive shell you can easily run most of the examples in this tutorial. Later, you will also learn how to easily include code written to a file as a module.

Type `python3` in the terminal:

```
$ python3
Python 3.10.4 (default, Mar 23 2022, 17:29:05)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Open a terminal window and enter `python3`:

```
$ python3
Python 3.10.4 (v3.10.4:9d38120e33, Mar 23 2022, 17:29:05) [Clang 13.0.0 (clang-1300.0.29.
↪30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Note: If you get the error message *Command not found*, you can run `Update Shell Profile`, which can be found in `Applications/Python3.10`.

You can start the interactive Python shell in *Start* → *Applications* → *Python 3.10*.

Alternatively, you can search for the directly executable file `Python.exe`, for example in `C:\Users\VEIT\AppData\Local\Programs\Python\Python310-64` and then double-click.

You can scroll through previous entries with the arrow keys `Home`, `End`, `Page up` and `Page down` and repeat with the `Enter` key.

3.1.1 Exiting the interactive shell

To exit the interactive shell, simply use `Ctrl-d` on Linux and macOS or `Ctrl-z` on Windows. Alternatively, you can type `exit()`.

3.2 IDLE

IDLE is the acronym for integrated development environment and combines an interactive interpreter with code editing and debugging tools. It is very easy to execute on the various platforms:

Enter the following into your terminal:

```
$ idle-python3.10
```

You can start IDLE in *Windows → All Apps → IDLE (Python GUI)*

You can scroll through the history of previous commands with the `alt-p` and `alt-n` keys.

CHAPTER

FOUR

EXPLORING PYTHON

Whether you use IDLE or the interactive shell, there are some useful functions to explore Python.

4.1 help()

`help()` has two different modes. When you type `help()`, you call the help system, which you can use to get information about modules, keywords, and other topics. When you are in the help system, you will see a prompt with `help>`. You can now enter a module name, for example `float`, to search the Python documentation for that type.

`help()` is part of the pydoc library, which provides access to the documentation built into Python libraries. Since every Python installation comes with full documentation, you have all the documentation at your fingertips even offline.

Alternatively, you can use `help()` more specifically by passing a type or variable name as a parameter, for example:

```
>>> x = 4.2
>>> help(x)
Help on float object:

class float(object)
 |  float(x=0, /)
 |
 |  Convert a string or number to a floating point number, if possible.
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
...
```

4.2 dir(), globals() and locals()

`dir()` is another useful function that lists objects in a specific *namespace*. If you use it without parameters, you can find out which methods and data are available locally. Alternatively, it can also list objects for a module or type.

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__',
↪'__spec__', 'x']
>>> dir(x)
['__abs__', '__add__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__
```

(continues on next page)

(continued from previous page)

```
↪divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__',
↪ '__ge__', '__getattribute__', '__getformat__', '__getnewargs__', '__getstate__', '__
↪gt__', '__hash__', '__init__', '__init_subclass__', '__int__', '__le__', '__lt__', '__
↪mod__', '__mul__', '__ne__', '__neg__', '__new__', '__pos__', '__pow__', '__radd__', '_
↪_rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '_
↪_rmul__', '__round__', '__rpow__', '__rsub__', '__rtruediv__', '__setattr__', '__
↪sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', 'as_
↪integer_ratio', 'conjugate', 'fromhex', 'hex', 'imag', 'is_integer', 'real']
```

In contrast to `dir()`, both `globals()` and `locals()` display the values associated with the objects. Currently, both functions return the same thing:

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class '_
↪frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {}, '__
↪builtins__': <module 'builtins' (built-in)>, 'x': 4.2}
```

CHAPTER

FIVE

STYLE

5.1 Indentation and blocks

Python differs from most other programming languages because it uses indentation to determine structure (that is, to determine what the *while* clause of a condition ETC. (et cetera) represents). Most other languages use curly braces to do this. In the following example, the indentation of lines 3–6 determines that they belong to the `while` statement:

```
1 >>> x, y = 6, 3
2 >>> while x > y:
3 ...     x -= 1
4 ...     if x == 4:
5 ...         break
6 ...     print(x)
```

Indentations to structure the code instead of curly braces takes a little getting used to, but offers significant advantages:

- You can have neither missing nor too many brackets. Also, you no longer have to search for the bracket that might match earlier brackets.
- The visual structure of the code reflects its actual structure, making it much easier to understand.
- Python coding styles are mostly uniform; in other words, your code will mostly look very similar to that of others.

5.2 Comments

Most of the time, anything that follows # is a comment and is ignored when the code is executed. The obvious exception is # in a *string*:

```
>>> x = "# This is a string and not a comment"
```

5.3 Basic Python style

In Python, there are relatively few restrictions on coding style, with the obvious exception that code must be divided into blocks by indentation. Even in this case, how (tabs or spaces) and how far indentation is used is not prescribed. However, there are preferred stylistic conventions for Python, which are contained in the *Python Enhancement Proposal* (PEP) 8. A selection of Python conventions can be found in the following table:

Context	Recommendation	Example
Module and package names	short, lower case, underscores only if necessary	`math`, `sys`
Function names	lower case, underscores if necessary	`my_func()`
Variable names	lower case, with underscores if necessary	`my_var`
Class names	CamelCase notation	`MyClass`
Constant names	Capital letters with underscores	`PI`
Indentation	Four spaces per level, no tabs	
Compare	not explicitly with `True` or `False`, see also *Boolean values and expressions*	`if my_var:`, `if not my_var:`

See also:

- **PEP 8**

I strongly recommend following the conventions of PEP 8. They are tried and tested, and make your code easier to understand for yourself and others.

CHAPTER

SIX

VARIABLES AND EXPRESSIONS

6.1 Variables

The most commonly used command in Python is assignment. The Python code to create a Vairiable called `x` that is to be given the value is:

```
>>> pi = 3.14159
```

In Python, unlike many other programming languages, neither a variable declaration nor an end-of-line delimiter is necessary. The line is terminated by the end of the line. Variables are created automatically when they are assigned for the first time.

Note: In Python, variables are labels that refer to objects. Any number of labels can refer to the same object, and if that object changes, so does the value to which all those variables refer. To better understand what this means, see the following example:

```
>>> x = [1, 2, 3]
>>> y = x
>>> y[0] = 4
>>> print(x)
[4, 2, 3]
```

However, variables can also refer to constants:

```
>>> x = 1
>>> y = x
>>> z = y
>>> y = 4
>>> print(x,y,z)
1 4 1
```

In this case, after the third line, `x`, `y` and `z` all refer to the same immutable integer object with the value `1`. The next line, `y = 4`, causes `y` to refer to the integer object `4`, but this does not change the references of `x` or `z`.

Python variables can be set to any object, whereas in many other languages variables can only be stored in the declared type.

Variable names are case-sensitive and can contain any alphanumeric character as well as underscores, but must begin with a letter or underscore.

Note: If you receive a `SyntaxError`, check whether the variable name is a keyword. Keywords are reserved for use in Python language constructs, so you cannot turn them into variables. After calling *help()* you can enter `keywords` to get the keywords:

```
>>> help()
...
help> keywords
Here is a list of the Python keywords.  Enter any keyword to get more help.
False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

Note: You can use a variable name to overwrite built-in functions, types and other objects so that they can then only be accessed via the builtins module. These variable names should therefore never be used. You can obtain a list of the `__builtins__` objects with :

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
↪'BaseExceptionGroup', 'BlockingIOError', 'BrokenPipeError', 'BufferError',
↪'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
↪'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning', 'EOFError',
↪'Ellipsis', 'EncodingWarning', 'EnvironmentError', 'Exception', 'ExceptionGroup',
↪'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning',
↪ 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',
↪'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
↪ 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
↪'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
↪'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
↪'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning
↪', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError
↪', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
↪'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
↪'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '__
↪build_class__', '__debug__', '__doc__', '__import__', '__loader__', '__name__', '__
↪package__', '__spec__', 'abs', 'aiter', 'all', 'anext', 'any', 'ascii', 'bin', 'bool',
↪'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile',
↪'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate',
↪'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals',
↪ 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
↪'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next',
↪'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr',
↪'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
↪ 'super', 'tuple', 'type', 'vars', 'zip']
```

6.2 Expressions

Python supports arithmetic and similar expressions. The following code calculates the average of `x` and `y` and stores the result in the variable `z`:

```
>>> x = 1
>>> y = 2
>>> z = (x + y) / 2
```

Note: Arithmetic operators that use only integers do not always return an integer. As of Python 3, division returns a floating point number. If you want the traditional integer division to return an integer, you can use `//` instead.

CHAPTER

SEVEN

DATA TYPES

Python has several built-in data types, such as *Numbers* (integers, floating point numbers, complex numbers), *strings*, *Lists*, *Tuples*, *Dictionaries*, *Sets* and *Files*. These data types can be manipulated using language operators, built-in functions, library functions or a data type's own methods.

You can also define your own classes and create your own class instances. For these class instances, you can define methods as well as manipulate them using the language operators and built-in functions for which you have defined the appropriate special method attributes.

Note: In the Python documentation and in this book, the term *object* is used for instances of any Python data type, not just what many other languages would call class instances. This is because all Python objects are instances of one class or another.

Python has several built-in data types, from scalars like numbers and boolean values to more complex structures like lists, dictionaries and files.

7.1 Numbers

Python's four number types are integers, floating point numbers, complex numbers and Boolean numbers:

Type	Examples
Integers	`-1`, `42`, `90000000`
Floats	`90000000.0`, `-0.005`, `9e7`, `-5e-3`
Complex numbers	`3 + 2j`, `-4- 2j`, `4.2 + 6.3j`
Boolean numbers	`True`, `False`

They can be manipulated with the arithmetic operators:

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`, `//`	Division[1]
`**`	Exponentiation
`%`	Modulus

[1] Dividing integers with `/` results in a float, and dividing integers with `//` results in an integer that is truncated.

Note: Integers can be unlimited in size, limited only by the available memory.

Examples:

```
>>> 8 + 3 - 5 * 3
-4
>>> 8 / 3
2.6666666666666665
>>> 8 // 3
2
>>> x = 4.2 ** 3.4
>>> x
131.53689544409096
>>> 9e7 * -5e-3
-450000.0
>>> -5e-3 ** 3
-1.2500000000000002e-07
```

See also:

- Julia Evans: Examples of floating point problems
- David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic

7.1.1 Complex numbers

Complex numbers consist of a real part and an imaginary part, which is given the suffix `j` in Python.

```
>>> 7 + 2j
(7+2j)
```

Note: Python expresses the resulting complex number in parentheses to indicate that the output represents the value of a single object:

```
>>> (5+3j) ** (3+5j)
(-7.04464115622119-11.276062812695923j)
```

```
>>> x = (5+3j) * (6+8j)
>>> x
(6+58j)
>>> x.real
6.0
>>> x.imag
58.0
```

Complex numbers consist of a real part and an imaginary part with the suffix `j`. In the preceding code, the variable `x` is assigned to a complex number. You can get its „real“ part with the attribute notation `x.real` and the „imaginary“ part with `x.imag`.

7.1.2 Built-in numerical functions

Several built-in functions can work with numbers:

`abs()`
: returns the absolute value of a number. Here, as argument can be an integer, a floating point number or an object that implements `__abs__()`. With complex numbers as arguments, their absolute value is returned.

`divmod()`
: takes two (non-complex) numbers as arguments and returns a pair of numbers consisting of their quotient and the remainder if integer division is used.

`float`
: returns a floating point number formed from a number or string `x`.

`hex()`
: converts an integer number to a lowercase hexadecimal string with the prefix `0x`.

`int`
: returns an integer object constructed from a number or string `x`, or `0` if no arguments are given.

`max()`
: returns the largest element in an iterable or the largest of two or more arguments.

`min()`
: returns the smallest element in an iterable or the smallest of two or more arguments.

`oct()`
: converts an integer number to an octal string with the prefix `0o`. The result is a valid Python expression. If `x` is not a Python `int()` object, it must define an `__index__()` method that returns an integer.

`pow()`
: returns *base* as a power of *exp*.

`round()`
: returns a number rounded to *ndigits* after the decimal point. If *ndigits* is omitted or is *None*, the nearest integer to the input is returned.

7.1.3 Boolean values

Boolean values are used in the following examples:

```
>>> x = False
>>> x
False
>>> not x
True
```

```
>>> y = True * 2
>>> y
2
```

Apart from their representation as `True` and `False`, Boolean values behave like the numbers `1` (`True`) and `0` (`False`).

7.1.4 Advanced numerical functions

More advanced numerical functions such as trigonometry, as well as some useful constants, are not built into Python, but are provided in a standard module called math. *Module* will be explained in more detail later. For now, suffice it to say that you need to make the maths functions available in this section by importing `math`:

```
import math
```

Built-in functions are always available and are called using standard function call syntax. In the following code, `round` is called with a float as the input argument.

```
>>> round(2.5)
2
```

With `ceil` from the standard library `math` and the attribute notation `MODUL.FUNKTION(ARGUMENT)` is rounded up:

```
>>> math.ceil(2.5)
3
```

The `math` module provides, among other things

- the number theoretic and representation functions `math.ceil()`, `math.modf()`, `math.frexp()` and `math.ldexp()`,
- the power and logarithmic functions `math.exp()`, `math.log()`, `math.log10()`, `math.pow()` and `math.sqrt()`,
- the trigonometric functions `math.acos()`, `math.asin()`, `math.atan()`, `math.atan2()`, `math.ceil()`, `math.cos()`, `math.hypot()` and `math.sin()`,
- the hyperbolic functions `math.cosh()`, `math.sinh()` and `math.tanh()`
- and the constants `math.e` and `math.pi`.

7.1.5 Advanced functions for complex numbers

The functions in the math module are not applicable to complex numbers; one of the reasons for this is probably that the square root of `-1` is supposed to produce an error. Therefore, similar functions for complex numbers have been provided in the cmath module:

`cmath.acos()`, `cmath.acosh()`, `cmath.asin()`, `cmath.asinh()`, `cmath.atan()`, `cmath.atanh()`, `cmath.cos()`, `cmath.cosh()`, `python3:cmath.e()`, `cmath.exp()`, `cmath.log()`, `cmath.log10()`, `python3:cmath.pi()`, `cmath.sin()`, `cmath.sinh()`, `cmath.sqrt()`, `cmath.tan()`, `cmath.tanh()`.

To make it clear in the code that these functions are special functions for complex numbers, and to avoid name conflicts with the more normal equivalents, it is recommended to simply import the module to explicitly refer to the `cmath` package when using the function, for example:

```
>>> import cmath
>>> cmath.sqrt(-2)
1.4142135623730951j
```

Warning: Now it becomes clearer why we do not recommend importing all functions of a module with `from MODULE import *`. If you would import the module `math` first and then the module `cmath`, the functions in `cmath` would have priority over those of `math`. Also, when understanding the code, it is much more tedious to find out the source of the functions used.

7.1.6 Rounding half to even

Usually Python calculates floating point numbers according to the IEEE 754 standard, rounding down numbers in the middle half of the time and rounding up in the other half to avoid statistical drift in longer calculations. `Decimal` and `ROUND_HALF_UP` from the decimal module are therefore needed for rounding half to even:

```
>>> import decimal
>>> num = decimal.Decimal("2.5")
>>> rounded = num.quantize(decimal.Decimal("0"), rounding = decimal.ROUND_HALF_UP)
>>> rounded
Decimal('3')
```

7.1.7 Numerical calculations

The standard Python installation is not well suited for intensive numerical calculations due to speed limitations. But the powerful Python extension NumPy provide highly efficient implementations of many advanced numerical operations. The focus is on array operations, including multi-dimensional matrices and advanced functions such as the fast Fourier transform.

7.1.8 Built-in modules for numbers

The Python standard library contains a number of built-in modules that you can use to manage numbers:

Module	Description
`numbers`	for numeric abstract base classes
`math`, `cmath`	for mathematical functions for real and complex numbers
`decimal`	for decimal fixed-point and floating-point arithmetic
`statistics`	for functions for calculating mathematical statistics
`fractions`	for rational numbers
`random`	for generating pseudo-random numbers and selections and for shuffling sequences
`itertools`	for functions that create iterators for efficient loops
`functools`	for higher-order functions and operations on callable objects
`operator`	for standard operators as functions

7.2 Lists

Python has a powerful built-in list type:

```
[]
[1]
[1, "2.", 3.0, ["4a", "4b"], (5.1,5.2)]
```

A list can contain a mixture of other types as elements, including strings, tuples, lists, dictionaries, functions, file objects and any kind of number.

A list can be indexed from the front or the back. You can also refer to a sub-segment of a list using slice notation:

```
>>> x = [1, "2.", 3.0, ["4a", "4b"], (5.1,5.2)]
>>> x[0]
'1'
>>> x[1]
'2.'
>>> x[-1]
(5.1, 5.2)
>>> x[-2]
['4a', '4b']
>>> x[1:-1]
['2.', 3.0, ['4a', '4b']]
>>> x[0:3]
[1, '2.', 3.0]
>>> x[:3]
[1, '2.', 3.0]
>>> x[-4:-1]
['2.', 3.0, ['4a', '4b']]
>>> x[-4:]
['2.', 3.0, ['4a', '4b'], (5.1, 5.2)]
```

Lines 2 and 4
Index from the beginning using positive indices starting with `0` as the first element.

Lines 6 and 8
Index from the back using negative indices starting with `-1` as the last element.

Lines 10 and 12
Slice with `[m:n]`, where `m` is the inclusive start point and `n` is the exclusive end point.

Lines 14, 16 and 18
A `[:n]` slice starts at the beginning and an `[m:]` slice goes to the end of a list.

You can use this notation to add, remove and replace elements in a list or to get an element or a new list that is a slice of it, for example:

```
>>> x = [1, "2.", 3.0, ["4a", "4b"], (5.1,5.2)]
>>> x[1] = "zweitens"
>>> x[2:3] = []
>>> x
[1, 'zweitens', ['4a', '4b'], (5.1, 5.2)]
>>> x[2] = [3.1, 3.2, 3.3]
>>> x
[1, 'zweitens', [3.1, 3.2, 3.3], (5.1, 5.2)]
>>> x[2:]
[[3.1, 3.2, 3.3], (5.1, 5.2)]
```

Line 3
The size of the list increases or decreases if the new slice is larger or smaller than the slice it replaces.

Slices also allow a step-by-step selection between the start and end indices. The default value for an unspecified stride is `1`, which takes every element from a sequence between the indices. With a stride of `2`, every second element is taken and so on:

```
>>> x[0:3:2]
[1, [3.1, 3.2, 3.3]]
```

(continues on next page)

(continued from previous page)

```
>>> x[::2]
[1, [3.1, 3.2, 3.3]]
>>> x[1::2]
['zweitens', (5.1, 5.2)]
```

The stride value can also be negative. A `-1` stride means counting from right to left:

```
>>> x[3:0:-2]
[(5.1, 5.2), 'zweitens']
>>> x[::-2]
[(5.1, 5.2), 'zweitens']
>>> x[::-1]
[(5.1, 5.2), [3.1, 3.2, 3.3], 'zweitens', 1]
```

Line 1
To use a negative increment, the start slice should be larger than the end slice.

Line 3
The exception is if you omit the start and end indices.

Line 5
A stride of `-1` reverses the order.

Some functions of the slice notation can also be executed with special operations, which improves the readability of the code:

```
>>> x.reverse()
>>> x
[(5.1, 5.2), [3.1, 3.2, 3.3], 'zweitens', 1]
```

You can also use the following built-in functions (`len`, `max` and `min`), some operators (`in`, `+` and `*`), the `del` statement and the list methods (`append`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse` and `sort`) for lists:

```
>>> len(x)
4
>>> x + [0, -1]
[(5.1, 5.2), [3.1, 3.2, 3.3], 'zweitens', 1, 0, -1]
>>> x.reverse()
>>> x
[1, 'zweitens', [3.1, 3.2, 3.3], (5.1, 5.2)]
```

Line 3
The operators + and * each create a new list, leaving the original list unchanged.

Line 5
The methods of a list are called using the attribute notation for the list itself: `:samp:`{LIST}.METHOD(ARGUMENTS)`.

Some of these operations repeat functions that can be performed using slice notation, but they improve the readability of the code.

See also:

- Select and filter data with pandas

7.2.1 Summary

data type	mutable	ordered	indexed	duplicates
list				

7.3 Tuples

Tuples are similar to lists but are immutable, so they cannot be changed once they have been created. The operators (`in`, `+` and `*`) and built-in functions (`len`, `max` and `min`) work with them in the same way as with lists, as none of these functions change the original. The index and slice notations work in the same way to get elements or slices, but cannot be used to add, remove or replace elements. Also, there are only two tuple methods: `count` and `index`. An important purpose of tuples is to be used as keys for dictionaries. They are also more efficient to use when you don't need a change facility.

```
1 ()
2 (1,)
3 (1, 2, 3, 5)
4 (1, "2.", 3.0, ["4a", "4b"], (5.1,5.2))
```

Line 2
A tuple with one element requires a comma.

Line 4
A tuple, like a *Liste*, can contain a mixture of other types as elements, including any *Numbers*, *Strings*, *Tuples*, *Lists*, *Dictionaries*, *Files* and functions.

A list can be converted to a tuple using the built-in `tuple` function:

```
>>> x = [1, 2, 3, 5]
>>> tuple(x)
(1, 2, 3, 5)
```

Conversely, a tuple can be converted into a list using the built-in list function:

```
>>> x = (1, 2, 3, 4)
>>> list(x)
[1, 2, 3, 4]
```

The advantages of tuples over *Lists* are:

- Tuples are faster than lists.

  If you want to define a constant set of values and just cycle through them, you should use a tuple instead of a list.

- Tuples can not be modified and are therefore *write-protected*.
- Tuples can be used as keys in *Dictionaries* and values in *Sets*.

7.3.1 Summary

data type	mutable	ordered	indexed	duplicates
tuple				

7.4 Sets

A set in Python is an unordered collection of objects used in situations where membership and uniqueness to the set are the most important information of the object. The `in` operator runs faster with sets than with *Lists*:

```
1 >>> x = set([1, 2, 3, 2, 4])
2 >>> x
3 {1, 2, 3, 4}
4 >>> 1 in x
5 True
6 >>> 5 in x
7 False
```

Line 1
You can create a set by applying `set` to a sequence like a *list*.

Line 3
When a sequence is made into a set, duplicates are removed.

Line 4 and 6
The keyword is used to check whether an object belongs to a set.

Sets behave like collections of *Dictionary* keys without associated values.

However, the speed advantage also comes at a price: sets do not keep the elements elements in the correct order, whereas *Lists* and *Tuples* do. If the order is important to you, you should use a data structure that remembers the order.

7.4.1 Summary

data type	mutable	ordered	indexed	duplicates
set				

7.5 Dictionaries

Python's built-in dictionary data type provides associative array functionality implemented using hash tables. The built-in `len` function returns the number of key-value pairs in a dictionary. The `del` statement can be used to delete a key-value pair. As with *Lists* , several dictionary methods (`clear`, `copy`, `get`, `items`, `keys`, `update` and `values`) are available.

```
>>> x = {1: "eins", 2: "zwei"}
>>> x[3] = "drei"
>>> x["viertes"] = "vier"
```

(continues on next page)

(continued from previous page)

```
>>> list(x.keys())
[1, 2, 3, 'viertes']
>>> x[1]
'eins'
>>> x.get(1, "nicht vorhanden")
'eins'
>>> x.get(5, "nicht vorhanden")
'nicht vorhanden'
```

Keys must be of immutable type, including *Numbers*, *Strings* and *Tuples*.

Warning: Even if you can use different key types in a dictionary, you should avoid this, as it not only makes it more difficult to read, but also sorting is also made more difficult.

Values can be any type of object, including mutable types such as *Lists* and *Dictionaries*. If you try to access the value of a key that is not in the dictionary, a `KeyError` exception is thrown. To avoid this error, the dictionary method `get` optionally returns a custom value if a key is not contained in a dictionary.

7.5.1 setdefault

`setdefault` can be used to provide counters for the keys of a dict, for example:

```
>>> titles = ["Data types", "Lists", "Sets", "Lists"]
>>> for title in titles:
...     titles_count.setdefault(title, 0)
...     titles_count[title] += 1
...
>>> titles_count
{'Data types': 1, 'Lists': 2, 'Sets': 1}
```

Note: Such counting operations quickly became widespread, so the `collections.Counter` class was later added to the Python standard library. This class can perform the above-mentioned operations much more easily:

```
>>> collections.Counter(titles)
Counter({'Lists': 2, 'Data types': 1, 'Sets': 1})
```

7.5.2 Merging dictionaries

You can merge two dictionaries into a single dictionary using the `dict.update()` method:

```
>>> titles = {7.0: "Data Types", 7.1: "Lists", 7.2: "Tuples"}
>>> new_titles = {7.0: "Data types", 7.3: "Sets"}
>>> titles.update(new_titles)
>>> titles
{7.0: 'Data types', 7.1: 'Lists', 7.2: 'Tuples', 7.3: 'Sets'}
```

Note: The order of the operands is important, as `7.0` is duplicated and the value of the last key overwrites the previous one.

7.5.3 Extensions

python-benedict
: `dict` subclass with keylist/keypath/keyattr support and I/O shortcuts.

pandas
: can convert Dicts into Series and DataFrames.

7.6 Strings

The processing of character strings is one of Python's strengths. There are many options for delimiting character strings:

```
"A string in double quotes can contain 'single quotes'."
'A string in single quotes can contain "double quotes"'
"'\tA string that starts with a tab and ends with a newline character.\n'"
"""This is a string in triple double quotes, the only string that contains
real line breaks."""
```

Strings can be separated by single (`' '`), double (`" "`), triple single (`''' '''`) or triple double (`""" """`) quotes and can contain tab (`\t`) and newline (`\n`) characters. In general, backslashes `\` can be used as escape characters. For example `\\` can be used for a single backslash and `\'` for a single quote, whereby it does not end the string:

```
"You don't need a backslash here."
'However, this wouldn\'t work without a backslash.'
```

Here are other characters you can get with the escape character:

Escape sequence	Output	Description
`\\`	\	Backslash
`\'`	'	single quote character
`\"`	"	double quote character
`\b`		Backspace (`BS`)
`\n`		ASCII Linefeed (`LF`)
`\r`		ASCII Carriage Return (`CR`)
`\t`		Tabulator (`TAB`)
`u00B5`	µ	Unicode 16 bit
`U000000B5`	µ	Unicode 32 bit
`N{SNAKE}`		Unicode Emoji name

A normal string cannot be split into multiple lines. The following code will not work:

```
"This is an incorrect attempt to insert a newline into a string without
using \n."
```

However, Python provides strings in triple quotes (`"""`) that allow this and can contain single and double quotes without backslashes.

Strings are also immutable. The operators and functions that work with them return new strings derived from the original. The operators (`in`, `+` and `*`) and built-in functions (`len`, `max` and `min`) work with strings in the same way as with lists and tuples.

```
>>> welcome = "Hello pythonistas!\n"
>>> 2 * welcome
'Hello pythonistas!\nHello pythonistas!\n'
>>> welcome + welcome
'Hello pythonistas!\nHello pythonistas!\n'
>>> 'python' in welcome
True
>>> max(welcome)
'y'
>>> min(welcome)
'\n'
```

The index and slice notation works in the same way to obtain elements or slices:

```
>>> welcome[0:5]
'Hello'
>>> welcome[6:-1]
'pythonistas!'
```

However, the index and slice notation cannot be used to add, remove or replace elements:

```
 >>> welcome[6:-1] = 'everybody!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

7.6.1 string

For strings, the standard Python library string contains several methods for working with their content, including `str.split()`, `str.replace()` and `str.strip()`:

```
>>> welcome = "hello pythonistas!\n"
>>> welcome.isupper()
False
>>> welcome.isalpha()
False
>>> welcome[0:5].isalpha()
True
>>> welcome.capitalize()
'Hello pythonistas!\n'
>>> welcome.title()
'Hello Pythonistas!\n'
 >>> welcome.strip()
'Hello pythonistas!'
>>> welcome.split(' ')
['hello', 'pythonistas!\n']
>>> chunks = [snippet.strip() for snippet in welcome.split(' ')]
>>> chunks
['hello', 'pythonistas!']
```

(continues on next page)

(continued from previous page)

```
>>> ' '.join(chunks)
'hello pythonistas!'
>>> welcome.replace('\n', '')
'hello pythonistas!'
```

Below you will find an overview of the most common string methods:

Method	Description
`str.count()`	returns the number of non-overlapping occurrences of the string.
`str.endswith()`	returns `True` if the string ends with the suffix.
`str.startswith()`	returns `True` if the string starts with the prefix.
`str.join()`	uses the string as a delimiter for concatenating a sequence of other strings.
`str.index()`	returns the position of the first character in the string if it was found in the string; triggers a `ValueError` if it was not found.
`str.find()`	returns the position of the first character of the first occurrence of the substring in the string; like `index`, but returns `-1` if nothing was found.
`str.rfind()`	Returns the position of the first character of the last occurrence of the substring in the string; returns `-1` if nothing was found.
`str.replace()`	replaces occurrences of a string with another string.
`str.strip()`, `str.rstrip()`, `str.lstrip()`	strip spaces, including line breaks.
`str.split()`	splits a string into a list of substrings using the passed separator.
`str.lower()`	converts alphabetic characters to lower case.
`str.upper()`	converts alphabetic characters to upper case.
`str.casefold()`	converts characters to lower case and converts all region-specific variable character combinations to a common comparable form.
`str.ljust()`, `str.rjust()`	left-aligned or right-aligned; fills the opposite side of the string with spaces (or another filler character) in order to obtain a character string with a minimum width.
`str.removeprefix()` `str.removesuffix()`	In Python 3.9 this can be used to extract the suffix or file name.

In addition, there are several methods with which the property of a character string can be checked:

Method	`[!#$%...]`	`[a-zA-Z]`	`[¼½¾]`	`[¹²³]`	`[0-9]`
`str.isprintable()`					
`str.isalnum()`					
`str.isnumeric()`					
`str.isdigit()`					
`str.isdecimal()`					

`str.isspace()` checks for spaces: `[ \t\n\r\f\v\x1c-\x1f\x85\xa0\u1680...]`.

7.6.2 re

The Python standard library re also contains functions for working with strings. However, `re` offers more sophisticated options for pattern extraction and replacement than `string`.

```
>>> import re
>>> re.sub('\n', '', welcome)
'Hello pythonistas!'
```

Here, the regular expression is first compiled and then its `re.Pattern.sub()` method is called for the passed text. You can compile the expression itself with `re.compile()` to create a reusable regex object that reduces CPU cycles when applied to different strings:

```
>>> regex = re.compile('\n')
>>> regex.sub('', welcome)
'Hello pythonistas!'
```

If you want to get a list of all patterns that match the `regex` object instead, you can use the `re.Pattern.findall()` method:

```
>>> regex.findall(welcome)
['\n']
```

Note: To avoid the awkward escaping with `\` in a regular expression, you can use raw string literals such as `r'C:\PATH\TO\FILE'` instead of the corresponding `'C:\\PATH\\TO\\FILE'`.

`re.Pattern.match()` and `re.Pattern.search()` are closely related to `re.Pattern.findall()`. While findall returns all matches in a string, `search` only returns the first match and `match` only returns matches at the beginning of the string. As a less trivial example, consider a block of text and a regular expression that can identify most email addresses:

```
>>> addresses = """Veit <veit@cusy.io>
... Veit Schiele <veit.schiele@cusy.io>
... cusy GmbH <info@cusy.io>
... """
>>> pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'
>>> regex = re.compile(pattern, flags=re.IGNORECASE)
>>> regex.findall(addresses)
['veit@cusy.io', 'veit.schiele@cusy.io', 'info@cusy.io']
>>> regex.search(addresses)
<re.Match object; span=(6, 18), match='veit@cusy.io'>
>>> print(regex.match(addresses))
None
```

`regex.match` returns `None`, as the pattern only matches if it is at the beginning of the string.

Suppose you want to find email addresses and at the same time split each address into its three components:

1. personal name
2. domain name
3. domain suffix

To do this, you first place round brackets `()` around the parts of the pattern to be segmented:

```
>>> pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
>>> regex = re.compile(pattern, flags=re.IGNORECASE)
>>> match = regex.match('veit@cusy.io')
>>> match.groups()
('veit', 'cusy', 'io')
```

`re.Match.groups()` returns a *Tuples* that contains all subgroups of the match.

`re.Pattern.findall()` returns a list of tuples if the pattern contains groups:

```
>>> regex.findall(addresses)
[('veit', 'cusy', 'io'), ('veit.schiele', 'cusy', 'io'), ('info', 'cusy', 'io')]
```

Groups can also be used in `re.Pattern.sub()` where `\1` stands for the first matching group, `\2` for the second and so on:

```
>>> regex.findall(addresses)
[('veit', 'cusy', 'io'), ('veit.schiele', 'cusy', 'io'), ('info', 'cusy', 'io')]
>>> print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', addresses))
Veit <Username: veit, Domain: cusy, Suffix: io>
Veit Schiele <Username: veit.schiele, Domain: cusy, Suffix: io>
cusy GmbH <Username: info, Domain: cusy, Suffix: io>
```

The following table contains a brief overview of methods for regular expressions:

Method	Description
`re.findall()`	returns all non-overlapping matching patterns in a string as a list.
`re.finditer()`	like `findall`, but returns an iterator.
`re.match()`	matches the pattern at the beginning of the string and optionally segments the pattern components into groups; if the pattern matches, a `match` object is returned, otherwise none.
`re.search()`	searches the string for matches to the pattern; in this case, returns a `match` object; unlike `match`, the match can be anywhere in the string and not just at the beginning.
`re.split()`	splits the string into parts each time the pattern occurs.
`re.sub()`, `re.subn()`	replaces all (`sub`) or the first `n` occurrences (`subn`) of the pattern in the string with a replacement expression; uses the symbols `\1`, `\2`, ... to refer to the elements of the match group.

See also:

- *Regular expressions*
- Regular Expression HOWTO
- re — Regular expression operations

7.6.3 print()

The function `print()` outputs character strings, whereby other Python data types can easily be converted into strings and formatted, for example:

```
>>> import math
>>> pi = math.pi
>>> d = 28
>>> u = pi * d
>>> print("Pi is", pi, "and the circumference with a diameter of", d, "inches is", u,
↪"inches.")
Pi is 3.141592653589793 and the circumference with a diameter of 28 inches is 87.
↪96459430051421 inches.
```

F-Strings

F-strings can be used to shorten numbers that are too detailed for a text:

```
>>> print(f"The value of Pi is {pi:.3f}.")
The value of Pi is 3.142.
```

In `{pi:.3f}`, the format specification `f` is used to truncate the number Pi to three decimal places.

In A/B test scenarios, you often want to display the percentage change in a key figure. F strings can be used to formulate them in an understandable way:

```
>>> metrics = 0.814172
>>> print(f"The AUC has increased to {metrics:=+7.2%}")
The AUC has increased to +81.42%
```

In this example, the variable `metrics` is formatted with = taking over the contents of the variable after the +, displaying a total of seven characters including the plus or minus sign, `metrics` and the percent sign. `.2` provides two decimal places, while the `%` symbol converts the decimal value into a percentage. For example, `0.514172` is converted to `+51.42%`.

Values can also be converted into binary and hexadecimal values:

```
>>> block_size = 192
>>> print(f"Binary block size: {block_size:b}")
Binary block size: 11000000
>>> print(f"Hex block size: {block_size:x}")
Hex block size: c0
```

There are also formatting specifications that are ideally suited for CLI (Command Line Interface) output, for example:

```
>>> data_types = [(7, "Data types", 19), (7.1, "Numbers", 19), (7.2, "Lists", 23)]
>>> for n, title, page in data_types:
...     print(f"{n:.1f} {title:.<25} {page: >3}")                                ...
7.0 Data types.............  19
7.1 Numbers................  19
7.2 Lists..................  23
```

In general, the format is as follows, whereby all information in square brackets is optional:

```
:[[FILL]ALIGN][SIGN][0b|0o|0x|d|n][0][WIDTH][GROUPING]["." PRECISION][TYPE]
```

The following table lists the fields for character string formatting and their meaning:

Field	Meaning
FILL	Character used to fill in ALIGN. The default value is a space.
ALIGN	Text alignment and fill character: <: left-aligned >: right-aligned ^: centred =: Fill character after SIGN
SIGN	Display sign: +: Display sign for positive and negative numbers -: Default value, - only for negative numbers or space for positive
0b|0o|0x|d|n	Sign for integers: 0b: Binary numbers 0o: Octal numbers 0x: Hexadecimal numbers d: Default value, decimal integer with base 10 n: uses the current locale setting to insert the corresponding number separators
0	fills with zeros
WIDTH	Minimum field width
GROUPING	Number separator:[1] ,: comma as thousands separator _: underscore for thousands separator
.PRECISION	 For floating point numbers, the number of digits after the point For non-numeric values, the maximum length
TYPE	Output format as number type or string ... for integers: b: binary format c: converts the integer to the corresponding Unicode character d: default value, decimal character n: same as d, th the difference that it uses the current locale setting to insert the corresponding number separators o: octal format x: Hexadecimal format in base 16, using lowercase letters for the digits above 9 X: Hexadecimal format based on 16, using capital letters for digits above 9 ... for floating point numbers: e: Exponent with e as separator between coefficient

Tip: A good source for F-strings is the help function:

```
>>> help()
help> FORMATTING
...
```

You can browse through the help here and find many examples.

You can exit the help function again with `:–q` and .

See also:

- PyFormat
- f-strings
- **PEP 498**

Debugging F-Strings

In Python 3.8, a specifier was introduced to help with debugging F-string variables. By adding an equals sign =, the code is included within the F-string:

```
>>> uid = "veit"
>>> print(f"My name is {uid.capitalize()=}")
My name is uid.capitalize()='Veit'
```

Formatting date and time formats and IP addresses

`datetime` supports the formatting of strings using the same syntax as the `strftime` method for these objects.

```
>>> import datetime
>>> today = datetime.date.today()
>>> print(f"Today is {today:%d %B %Y}.")
Today is 26 November 2023.
```

The `ipaddress` module of Python also supports the formatting of `IPv4Address` and `IPv6Address` objects.

Finally, third-party libraries can also add their own support for formatting strings by adding a `__format__` method to their objects.

See also:

- strftime() and strptime() Format Codes
- Python strftime cheatsheet

[1] The format identifier `n` formats a number in a locally customised way, for example:

```
>>> value = 635372
>>> import locale
>>> locale.setlocale(locale.LC_NUMERIC, "en_US.utf-8")
'en_US.utf-8'
>>> print(f"{value:n}")
635,372
```

7.6.4 Built-in modules for strings

The Python standard library contains a number of built-in modules that you can use to manage strings:

Module	Description
`string`	compares with constants such as `string.digits` or `string.whitespace`
`re`	searches and replaces text with regular expressions
`struct`	interprets bytes as packed binary data
`difflib`	helps to calculate deltas, find differences between strings or sequences and create patches and diff files
`textwrap`	wraps and fills text, formats text with line breaks or spaces

See also:

- Manipulation of strings with pandas

7.7 Files

7.7.1 Opening files

In Python, you open and read a file using the built-in `open()` function and various built-in read operations. The following short Python program reads a line from a text file called *`myfile.txt`*:

```
>>> f = open("docs/types/myfile.txt", "r")
>>> line = f.readline()
```

`open()` does not read anything from the file, but returns a so-called file object that you can use to access the open file. It keeps track of a file and how much of the file has been read or written. All file input in Python is done with file objects, not file names.

The first call to `readline` returns the first line of the file object, which is everything up to and including the first line break, or the entire file if there is no line break in the file; the next call to `readline` returns the second line if it exists, and so on.

The first argument of the `open` function is a pathname. In the previous example, you open a file that you assume is in the current working directory. The following example opens a file in an absolute location – *`C:\My Documents\myfile`*:

```
>>> import os
>>> pathname = os.path.join("C:/", "Users", "Veit", "Documents", "myfile.txt")
>>> with open(pathname, "r") as f:
...     line = f.readline()
```

Note: This example uses the `with` keyword, which means that the file is opened with a context manager, which is explained in more detail in *Context management with with*. This way of opening files manages possible I/O errors better and should generally be preferred.

7.7.2 Closing files

After all data has been read from or written to a file object, the file object should be closed again to free up system resources, allow other code to read or write to the underlying file, and make the program more reliable overall. For small scripts, this usually does not have a large impact because file objects are automatically closed when the script or program exits. However, for larger programs, too many open file objects can exhaust system resources, causing the program to terminate. You close a file object with the `close` method when the file object is no longer needed:

```
>>> f = open("docs/types/myfile.txt", "r")
>>> line = f.readline()
>>> f.close()
```

However, using a *Context management with with* usually remains the better option to automatically close files when you are done:

```
>>> with open("docs/types/myfile.txt", "r") as f:
...     line = f.readline()
```

7.7.3 Opening files in write or other modes

The second argument of the `open()` function is a string that specifies how the file should be opened. `"r"` opens the file for reading, `"w"` opens the file for writing, and `"a"` opens the file for attaching. If you want to open the file for reading, you can omit the second argument, because `"r"` is the default value. The following short program writes `Hi, Pythonistas!` to a file:

```
>>> f = open("docs/types/myfile.txt", "w")
>>> f.write("Hi, Pythonistas!\n")
17
>>> f.close()
```

Depending on the operating system, `open()` may also have access to other file modes. However, these modes are not necessary for most purposes.

`open` can take an optional third argument that defines how read or write operations for this file are buffered. Buffering keeps data in memory until enough data has been requested or written to justify the time required for a disk access. Other parameters for `open` control the encoding for text files and the handling of line breaks in text files. Again, you don't usually need to worry about these functions, but as you become more advanced with Python you may want to read up on them.

7.7.4 Read and write functions

I have already introduced the most common function for reading text files, `readline`. This function reads a single line from a file object and returns it, including all line breaks at the end of the line. If there is nothing more to read, readline returns an empty string, which makes it easy to determine, for example, the number of lines in a file:

```
>>> f = open("docs/types/myfile.txt", "r")
>>> lc = 0
>>> while f.readline() != "":
...     lc = lc + 1
...
>>> print(lc)
1
>>> f.close()
```

A shorter way to count all lines is with the `readlines` method, which is also built in, that reads all lines of a file and returns them as a list of strings with one string per line:

```
>>> f = open("docs/types/myfile.txt", "r")
>>> print(len(f.readlines()))
1
>>> f.close()
```

If you count all the lines in a large file, this method may cause the memory to fill up because the entire file is read at once. It is also possible that memory overflows with `readline` if you try to read a line from a large file that does not contain newline characters. To better deal with such situations, both methods have an optional argument that affects the amount of data read at a time. Another way to iterate over all the lines in a file is to treat the file object as an iterator in a *for loop*:

```
>>> f = open("docs/types/myfile.txt", "r")
>>> lc = 0
>>> for l in f:
...     lc = lc + 1
...
>>> print(lc)
1
>>> f.close()
```

This method has the advantage that the lines are read into the memory as needed, so that even with large files there is no need to fear a lack of memory. The other advantage of this method is that it is simpler and more readable.

However, a possible problem with the read method can arise when conversions are done in text mode on Windows and macOS if you use the `open()` command in text mode, that is without appending a `b`. In text mode on macOS, each `\r` is converted to `\n`, while on Windows, `\r\n` pairs are converted to `\n`. You can specify how line breaks are handled by using the `newline` parameter when opening the file and specifying `newline="\n"`, `\r` or `\r\n`, which will cause only that string to be used as a line break:

```
>>> f = open("docs/types/myfile.txt", "r", newline="\n")
```

In this example, only `\n` is considered a line break. However, if the file was opened in binary mode, the `newline` parameter is not necessary, as all bytes are returned exactly as they are in the file.

The write methods corresponding to `readline` and `readlines` are `write` and `writelines`. Note that there is no `writeline` function. `write` writes a single string that can span multiple lines if newline characters are embedded in the string, as in the following example:

```
f.write("Hi, Pythinistas!\n\n")
```

The `writelines` method is confusing, however, because it does not necessarily write multiple lines; it takes a list of strings as an argument and writes them sequentially to the specified file object without inserting line breaks between the list items; only if the strings in the list contain line breaks are line breaks added to the file object; otherwise they are concatenated. `writelines` is thus the exact inverse of `readlines`, since it can be applied to the list returned by `readlines` to write a file identical to the source file. Assuming that `myfile.txt` exists and is a text file, the following example creates an exact copy of `myfile.txt` named `myfile2.txt`:

```
>>> input_file = open("myfile.txt", "r")
>>> lines = input_file.readlines()
>>> input_file.close()
>>> output_file = open("myfile2.txt", "w")
>>> output_file.writelines(lines)
>>> output_file.close()
```

Using binary mode

If you want to read all the data in a file (partially) into a single byte object and transfer it to memory to be treated as a byte sequence, you can use the `read` method. Without an argument, it reads the entire file from the current position and returns the data as a byte object. With an integer argument, it reads a maximum of this number of bytes and returns a bytes object of the specified size:

```
1 >>> f = open("myfile.txt", "rb")
2 >>> head = f.read(16)
3 >>> print(head)
4 b'Hi, Pythonistas!'
5 >>> body = f.read()
6 >>> print(body)
7 b'\n\n'
8 >>> f.close()
```

Line 1
opens a file for reading in binary mode

Line 2
reads the first 16 bytes as `head` string

Line 3
outputs the `head` string

Line 5
reads the rest of the file

Note: Files opened in binary mode work only with bytes and not with strings. To use the data as strings, you must decode all byte objects into string objects. This point is often important when dealing with network protocols, where data streams often behave like files, but must be interpreted as bytes and not strings.

7.7.5 Built-in modules for files

The Python standard library contains a number of built-in modules that you can use to manage files:

Module	Description
os.path	performs common pathname manipulations
pathlib	manipulates pathnames
fileinput	iterates over multiple input files
filecmp	compares files and directories
tempfile	creates temporary files and directories
glob, fnmatch	use UNIX-like path and file name patterns
linecache	randomly accesses lines of text
shutil	performs higher level file operations
mimetypes	Assignment of file names to MIME types
pickle, shelve	enable Python object serialisation and persistence, see also *The pickle module*
csv	reads and writes CSV files
json	JSON encoder and decoder
sqlite3	provides a DB-API 2.0 interface for SQLite databases, see also *The sqlite module*
xml, xml.parsers.expat, xml.dom, xml.sax, xml.etree.ElementTree	reads and writes XML files, see also R:doc:*../save-data/xml*
html.parser, html.entities	Parsing HTML and XHTML
configparser	reads and writes Windows-like configuration files (.ini)
base64, binhex, binascii, quopri, uu	encodes/decodes files or streams
struct	reads and writes structured data to and from files
zlib, gzip, bz2, zipfile, tarfile	for working with archive files and compressions

See also:

- pandas IO tools
- Examples of serialisation formats CSV, JSON, Excel, XML/HTML, YAML, TOML und Pickle.

7.8 None

In addition to the standard types such as *Strings* and *Numbers*, Python has a special data type that defines a single special data object called `None`. As the name suggests, `None` is used to represent an empty value. It appears in various forms in Python.

`None` is often useful in everyday Python programming as a placeholder to indicate a data structure where meaningful data can eventually be found, even if that data has not yet been calculated.

The presence of `None` is easy to check, as there is only one instance of `None` in Python (all references to `None` point to the same object), and `None` is only identical to itself:

```
>>> MyType = type(None)
>>> MyType() is None
True
```

7.8.1 None is falsy

In Python, we often rely on the fact that None is falsy:

```
>>> bool(None)
False
```

For example, we can check whether *Strings* are empty in an *if statement*:

```
>>> myval = ""
>>> if not myval:
...     print("No value was specified.")
...
No value was specified.
```

7.8.2 None stands for emptiness

```
>>> titles = {7.0: "Data Types", 7.1: "Lists", 7.2: "Tuples"}
>>> third_title = titles.get("7.3")
>>> print(third_title)
None
```

7.8.3 The default return value of a function is None

For example, a procedure in Python is just a function that does not explicitly return a value, which means that it returns None by default:

```
>>> def myfunc():
...     pass
...
>>> print(myfunc())
None
```

CHAPTER

EIGHT

INPUT

You can use the `input()` function to get data input. Use the prompt string you want to display as a parameter for `input`:

```
>>> first_name = input("First name? ")
First name? Veit
>>> surname = input("Surname? ")
Surname? Schiele
>>> print(first_name, surname)
Veit Schiele
```

This is a fairly simple way to get data input. The only catch is that the input comes in as a string. So if you want to use a number, you have to convert it with the `int` or `float` function, for example, for calculating the age from the year of birth:

```
>>> import datetime
>>>
>>> currentDateTime = datetime.datetime.now()
>>> year = currentDateTime.year
>>> year_birth = input("Year of birth? ")
Year of birth? 1964
>>> age = year - int(year_birth)
>>> print('Age:', age, 'years')
Age: 58 years
```

CHAPTER

NINE

CONTROL FLOWS

Python has a whole range of structures for controlling code execution and programme flow, including common branches and loops.

9.1 Boolean values and expressions

In Python, there are several ways to express Boolean values; the Boolean constant `False`, `0`, the Python value `None`, and empty values (for example, the empty list `[]` or the empty string `""`) are all considered `False`. The Boolean constant `True` and everything else is considered `True`.

<, <=, ==, >, >=
: compares values.

is, is not, in, not in
: checks the identity.

and, not, or
: are logical operators that can be used to link the above checks.

```
>>> x = 3
>>> y = 3.0
>>> z = [3, 4, 5]
>>> x == y
True
>>> x is y
False
>>> x is not y
True
>>> x in z
True
>>> id(x)
4375911432
>>> id(y)
4367574480
>>> id(z[0])
4375911432
```

If `x` and `z[0]` have the same ID in memory, this means that we are referring to the same object in two places.

Most frequently, `is` and `is not` are used in conjunction with *None*:

```
>>> x is None
False
>>> x is not None
True
```

The Python style guide in **PEP 8** says that you should use identity to compare with *None*. So you should never use `x == None`, but enter `x is None` instead.

However, you should never compare calculated floating point numbers with each other:

```
>>> u = 0.6 * 7
>>> v = 0.7 * 6
>>> u == v
False
>>> u
4.2
>>> v
4.199999999999999
```

9.2 `if-elif-else` statement

The code block after the first true condition of an `if` or `elif` statement is executed. If none of the conditions are true, the code block after the `else` is executed:

```
1  >>> x = 1
2  >>> if x < 1:
3  ...     x = 2
4  ...     y = 3
5  ... elif x > 1:
6  ...     x = 4
7  ...     y = 5
8  ... else:
9  ...     x = 6
10 ...     y = 7
11 ...
12 >>> print(x, y)
13 6 7
```

Lines 5 and 8
: The `elif` and `else` clauses are optional, and there can be any number of `elif` clauses.

Lines 3, 4, 6, 7, 9 and 10
: Python uses indentations to delimit blocks. No explicit delimiters such as brackets or curly braces are required. Each block consists of one or more statements separated by line breaks. All these statements must be on the same indentation level.

9.3 Loops

9.3.1 `while` loop

The `while` loop is executed as long as the condition (here: `x > y`) is true:

```
>>> x, y = 6, 3
>>> while x > y:
...     x -= 1
...     if x == 4:
...         break
...     print(x)
...
5
```

Line 1
This is a shorthand notation where `x` is given the value `6` and `y` is given the value `3`.

Lines 2–10
This is the `while` loop with the statement `x > y`, which is true as long as `x` is greater than `y`.

Line 3
`x` is reduced by `1`.

Line 4
`if` condition where `x` is to be exactly `4`.

Line 5
`break` ends the loop.

Lines 8 and 9
outputs the results of the `while` loop before execution was interrupted with `break`.

```
>>> x, y = 6, 3
>>> while x > y:
...     x -= 1
...     if x == 4:
...         continue
...     print(x)
...
5
3
```

Line 5
`continue` terminates the current iteration of the loop.

9.3.2 `for` loop

The `for` loop is simple but powerful because it can iterate over any iterable type, such as a list or a tuple. Unlike many other languages, the `for` loop in Python iterates over every element in a sequence for example a *list* or a *tuple*), which makes it more like a foreach loop. The following loop uses the Modulo operator `%` as a condition for the first occurrence of an integer divisible by `5`:

```
>>> items = [1, "fünf", 5.0, 10, 11, 15]
>>> d = 5
>>> for i in items:
...     if not isinstance(i, int):
...         continue
...     if not i % d:
...         print(f"First integer found that is divisible by {d}: {i}")
...         break
...
First integer found that is divisible by 5: 10
```

`x` is assigned each value in the list in turn. If `x` is not an integer, the remainder of this iteration is aborted by the `continue` statement. The flow control is continued with `x` being set to the next entry in the list. After the first matching integer is found, the loop is terminated with the `break` statement.

9.3.3 Loops with an index

You can also output the index in a `for` loop, for example with `enumerate()`:

```
>>> data_types = ["Data types", "Numbers", "Lists"]
>>> for index, title in enumerate(data_types):
...     print(index, title)
...
0 Data types
1 Numbers
2 Lists
```

9.3.4 List Comprehensions

A list is usually generated as follows:

```
>>> squares = []
>>> for i in range(8):
...     squares.append(i ** 2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49]
```

Instead of creating an empty list and inserting each element at the end, with list comprehensions you simply define the list and its content at the same time with just a single line of code:

```
>>> squares = [i ** 2 for i in range(8)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49]
```

The general format for this is:

NEW_LIST = [*EXPRESSION* for *MEMBER* in *ITERABLE*]

Each list comprehension in Python contains three elements:

EXPRESSION
: is a call to a method or another valid expression that returns a value. In the example above, the expression `i ** 2` is the square of the respective member value.

MEMBER
: is the object or the value in an *ITERABLE*. In the example above, the value is `i`.

ITERABLE
: is a *list*, a *set*, a generator or another object that can return its elements individually. In the example above, the iterable is `range(8)`.

You can also use optional conditions with list comprehensions, which are usually appended to the end of the expression:

```
>>> squares = [i ** 2 for i in range(8) if i >= 4]
>>> squares
[16, 25, 36, 49]
```

9.4 Exceptions

This section is about exceptions, that is, language functions that specifically handle unusual circumstances during the execution of a programme. The most common exception is to handle errors, but they can also be used effectively for many other purposes. Python provides a comprehensive set of exceptions, and you can define new exceptions for your own purposes.

The entire exception mechanism in Python is object-oriented: An exception is an object that is automatically created by Python functions with a `raise` statement. This `raise` statement causes the Python programme to be executed in a different way than usually intended: The current call chain is searched for a handler that can handle the generated exception. If such a handler is found, it is called and can access the exception object to obtain further information. If no suitable exception handler is found, the programme terminates with an error message.

It is possible to create different types of exceptions to reflect the actual cause of the reported error or unusual circumstance. For an overview of the class hierarchy of built-in exceptions, see Exception hierarchy in the Python documentation. Each exception type is a Python class that inherits from its parent exception type. For example, a `ZeroDivisionError` is also an `ArithmeticError`, an `Exception` and also a `BaseException` by inheritance. This hierarchy is intentional: most exceptions inherit from `Exception`, and it is strongly recommended that all user-defined exceptions also subclass `Exception`, and not `BaseException`:

```
class EmptyFileError(Exception):
    pass
```

This defines your own exception type, which inherits from the `Exception` base type.

```
filenames = ["myFile1.py", "nonExistent.py", "emptyFile.py", "myFile2.py"]
```

A list of different file types is defined.

Finally, exceptions or errors are caught and handled using the compound statement `try-except-else-finally`. Any exception that is not caught will cause the programme to terminate.

```
7     try:
8         f = open(file, "r")
9         line = f.readline()
10        if line == "":
11            f.close()
12            raise EmptyFileError(f"{file} is empty")
13    except IOError as error:
14        print(f"Cannot open file {file}: {error.strerror}")
15    except EmptyFileError as error:
16        print(error)
17    else:
18        print(f"{file}: {f.readline()}")
19    finally:
20        print("File", file, "processed")
```

Line 7
If an `IOError` or `EmptyFileError` occurs during the execution of the instructions in the `try` block, the corresponding `except` block is executed.

Line 9
An `IOError` could be triggered here.

Line 12
Here you trigger the `EmptyFileError`.

Line 17
The `else` clause is optional; it is executed if no exception occurs in the `try` block.

Note: In this example, `continue` statements could have been used in the `except` blocks instead.

Line 19
The `finally` clause is optional; it is executed at the end of the block, regardless of whether an exception was thrown or not.

9.5 Context management with `with`

A more rational way to encapsulate the `try-except-finally` pattern is to use the keyword `with` and a context manager. Python defines context managers for things like *file* access and custom context managers. One advantage of context managers is that they can define default clean-up actions that are always executed, whether an exception occurs or not.

The following listing shows opening and reading a file using `with` and a context manager.

```
1 filename = "myFile1.py"
2 with open(filename, "r") as f:
3     for line in f:
4         print(f)
```

A context manager is set up here that encloses the `open` function and the block that follows it. The predefined clean-up action of the context manager closes the file even if an exception occurs. As long as the expression in the first line is executed without throwing an exception, the file is always closed. This code is equivalent to this code:

```
filename = "myfile1.py"
try:
    f = open(filename, "r")
    for line in f:
        print(f)
except Exception as e:
    raise e
finally:
    f.close()
```

CHAPTER

TEN

FUNCTIONS

10.1 Basic function definitions

The basic syntax for a Python function definition is

```
def function_name(param1, param2, ...):
    body
```

As with *control streams*, Python uses indentation to separate the function from the function definition. The following simple example inserts the code into a function so that you can call it to get the factorial of a number:

```
1 >>> def fact(n):
2 ...     """Return the factorial of the given number."""
3 ...     f = 1
4 ...     while n > 0:
5 ...         f = f * n
6 ...         n = n - 1
7 ...     return f
```

Line 2

This is an optional documentation string, or `docstring`. You can get its value by calling `fact.__doc__`. The purpose of docstrings is to describe the behaviour of a function and the parameters it takes, while comments are to document internal information about how the code works. Docstrings are *Strings* that immediately follow the first line of a function definition and are usually enclosed in triple quotes to allow for multi-line descriptions. For multi-line documentation strings, it is common to give a summary of the function on the first line, follow this summary with an empty line and end with the rest of the information.

See also:

- *sphinx.ext.napoleon*

Line 7

The value is returned after the function is called. You can also write functions that have no return statement and return *None*, and when `return arg` is executed, the value `arg` is returned.

Although all Python functions return values, it is up to you how the return value of a function is used:

```
1 >>> fact(3)
2 6
3 >>> x = fact(3)
4 >>> x
5 6
```

Line 1
The return value is not linked to a variable.

Line 2
The value of the `fact` function is only output in the interpreter.

Line 3
The return value is linked to the variable `x`.

10.2 Parameters

Python offers flexible mechanisms for passing arguments to functions:

```
>>> x, y = 2, 3
>>> def func1(u, v, w):
...     value = u + 2*v + w**2
...     if value > 0:
...         return u + 2*v + w**2
...     else:
...         return 0
...
>>> func1(x, y, 2)
12
>>> func1(x, w=y, v=2)
15
>>> def func2(u, v=1, w=1):
...     return u + 4 * v + w ** 2
...
>>> func2(5, w=6)
45
>>> def func3(u, v=1, w=1, *tup):
...     print((u, v, w) + tup)
...
>>> func3(7)
(7, 1, 1)
>>> func3(1,2,3,4,5)
(1, 2, 3, 4, 5)
>>> def func4(u, v=1, w=1, **kwargs):
...     print(u, v, w, kwargs)
...
>>> func4(1, 2, s=4, t=5, w=3)
1 2 3 {'s': 4, 't': 5}
```

Line 2
Functions are defined with the `def` statement.

Line 5
The `return` statement is used by a function to return a value. This value can be of any type. If no `return` statement is found, the value `None` is returned by Python.

Line 11
Function arguments can be entered either by position or by name (keyword). `z` and `y` are specified by name in our example.

Line 13
Function parameters can be defined with default values that will be used if a function call omits them.

Line 18
A special parameter can be defined that combines all additional positional arguments in a function call into one tuple.

Zeile 25
Similarly, a special parameter can be defined that summarises all additional keyword arguments in a function call in a dictionary.

10.2.1 Parameters

Options for function parameters

Most functions need parameters. Python offers three options for defining function parameters.

Positional parameters

The simplest way to pass parameters to a function in Python is to pass them at the position. On the first line of the function, you specify the variable name for each parameter; when the function is called, the parameters used in the calling code are assigned to the function's parameter variables based on their order. The following function calculates `x` as a power of `y`:

```
>>> def power(x, y):
...     p = 1
...     while y > 0:
...             p = p * x
...             y = y - 1
...     return p
...
>>> power(2, 5)
32
```

This method requires that the number of parameters used by the calling code exactly matches the number of parameters in the function definition; otherwise, a type error exception is thrown:

```
>>> power(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() missing 1 required positional argument: 'y'
```

Function parameters can have default values, which you can declare by assigning a default value in the first line of the function definition, like this:

```
def function_name(param1, param2=Standardwert2, param3=Standardwert3, ...)
```

Any number of parameters can be given default values, but parameters with default values must be defined as the last in the parameter list.

The following function also calculates `x` as a power of `y`. However, if `y` is not specified in a function call, the default value 5 is used:

```
>>> def power(x, y=5):
...     p = 1
...     while y > 0:
...             p = p * x
...             y = y - 1
...     return p
```

You can see the effect of the standard argument in the following example:

```
>>> power(3, 6)
729
>>> power(3)
243
```

Parameter names

You can also pass arguments to a function by using the name of the corresponding function parameter rather than its position. Similar to the previous example, you can enter the following:

```
>>> power(y=6, x=2)
64
```

Since the arguments for the power are named `x` and `y` in the last call, their order is irrelevant; the arguments are linked to the parameters of the same name in the definition of the power, and you get back `2^6`. This type of argument passing is called keyword passing. Keyword passing can be very useful in combination with the default arguments of Python functions when you define functions with a large number of possible arguments, most of which have common default values.

Variable number of arguments

Python functions can also be defined to handle a variable number of arguments. This is possible in two ways. One method collects an unknown number of arguments in a *list*. The other method can collect an arbitrary number of arguments passed with a keyword that has no correspondingly named parameter in the function parameter list in a *dict*.

For an indeterminate number of positional arguments, prefixing the function's final parameter name with a * causes all excess non-keyword arguments in a function call, that is, the positional arguments that are not assigned to any other parameter, to be collected and assigned as a tuple to the specified parameter. This is, for example, a simple way to implement a function that finds the mean in a list of numbers:

```
>>> def mean(*numbers):
...     if len(numbers) == 0:
...         return None
...     else:
...         m = sum(numbers) / len(numbers)
...     return m
```

Now you can test the behaviour of the function, for example with:

```
>>> mean(3, 5, 2, 4, 6)
4.0
```

Any number of keyword arguments can also be processed if the last parameter in the parameter list is prefixed with **. Then all arguments passed with a keyword are collected in a *dict*. The key for each entry in the dict is the keyword (parameter name) for the argument. The value of this entry is the argument itself. An argument passed by keyword is superfluous in this context if the keyword with which it was passed does not match one of the parameter names in the function definition, for example:

```
>>> def server(ip, port, **other):
...     print("ip: {0}, port: {1}, keys in 'other': {2}".format(ip,
...           port, list(other.keys())))
...     total = 0
...     for k in other.keys():
...         total = total + other[k]
...     print("The sum of the other values is {0}".format(total))
```

Trying out this function shows that it can add the arguments passed under the keywords `foo`, `bar` and `baz`, even though `foo`, `bar` and `baz` are not parameter names in the function definition:

```
>>> server("127.0.0.1", port = "8080", foo = 3, bar = 5, baz = 2)
ip: 127.0.0.1, port: 8080, keys in 'other': ['foo', 'bar', 'baz']
The sum of the other values is 10
```

Mixing argument passing techniques

It is possible to use all the argument passing techniques of Python functions at the same time, although this can be confusing if you don't do it carefully. Positional arguments should come first, then named arguments, followed by indefinite positional arguments with a simple *, and finally indefinite keyword arguments with **.

Mutable objects as arguments

Arguments are passed by object reference. The parameter becomes a new reference to the object. With immutable objects such as *Tuples*, *Strings* and *Numbers*, what is done with a parameter has no effect outside the function. However, if you pass a mutable object, such as a *Lists*, a *Dictionaries* or a class instance, any change to the object changes what the argument refers to outside the function. Reassigning the parameter has no effect on the argument.

```
>>> def my_func(n, l):
...     l.append(1)
...     n = n + 1
...
>>> x = 5
>>> y = [2, 4, 6]
>>> my_func(x, y)
>>> x, y
(5, [2, 4, 6, 1])
```

The variable `x` is not changed because it is unchangeable. Instead, the function parameter `n` is set so that it refers to the new value 6. However, there is a change in `y` because the list it refers to has been changed.

10.2.2 Variables

Local, non-local and global variables

Here you return to the definition of `fact` from the beginning of this *Functions* chapter:

```
>>> def fact(n):
...     """Return the factorial of the given number."""
...     f = 1
...     while n > 0:
...         f = f * n
...         n = n - 1
...     return f
```

Both the variables `f` and `n` are local to a particular call to the function `fact`; changes made to them during the execution of the function have no effect on variables outside the function. All variables in the parameter list of a function and all variables created within a function by an assignment, such as `f = 1`, are local to the function.

You can explicitly make a variable a global variable by declaring it with the `global` statement before it is used. Global variables can be accessed and changed by the function. They exist outside the function and can also be accessed and changed by other functions that declare them as global, or by code that is not inside a function. Here is an example that illustrates the difference between local and global variables:

```
>>> def my_func():
...     global x
...     x = 1
...     y = 2
```

```
>>> x = 3
>>> y = 4
>>> my_func()
>>> x
1
>>> y
4
```

In this example, a function is defined that treats `x` as a global variable and `y` as a local variable, and attempts to change both `x` and `y`. The assignment to `x` within `my_func` is an assignment to the global variable `x`, which also exists outside `my_func`. Since `x` is designated as global in `my_func`, the assignment changes this global variable so that it retains the value `1` instead of the value `3`. However, the same is not true for `y`; the local variable `y` inside `my_func` initially refers to the same value as the variable `y` outside `my_func`, but the assignment causes `y` to refer to a new value that is local to the `my_func` function.

See also:

- The global statement

While `global` is used for a top-level variable, `nonlocal` refers to any variable in an enclosing area.

See also:

- The nonlocal statement
- **PEP 3104**

10.2.3 Decorators

Functions can also be passed as arguments to other functions and return the results of other functions. For example, it is possible to write a Python function that takes another function as a parameter, embeds it in another function that does something similar, and then returns the new function. This new combination can then be used instead of the original function:

```
>>> def inf(func):
...     print("Information about", func.__name__)
...     def details(*args):
...         print("Execute function", func.__name__, "with the argument(s)")
...         return func(*args)
...     return details
...
>>> def my_func(*params):
...     print(params)
...
>>> my_func = inf(my_func)
Information about my_func
>>> my_func("Hello", "Pythonistas!")
Execute function my_func with the argument(s)
('Hello', 'Pythonistas!')
```

Line 2
The `inf` function outputs the name of the function it wraps.

Line 6
When finished, the `inf` function returns the wrapped function.

A decorator is syntactic sugar for this process and allows you to wrap one function inside another with a one-line addition. You still get exactly the same effect as with the previous code, but the resulting code is much cleaner and easier to read. Using a decorator simply consists of two parts:

1. the definition of the function to wrap or *decorate* other functions, and
2. the use of an `@` followed by the decorator just before the wrapped function is defined.

The decorator function should take a function as a parameter and return a function, as follows:

```
>>> @inf
... def my_func(*params):
...     print(params)
...
Information about my_func
>>> my_func("Hello", "Pythonistas!")
Execute function my_func with the argument(s)
('Hello', 'Pythonistas!')
```

Line 1
The function `my_func` is decorated with `@inf`.

Line 7
The wrapped function is called after the decorator function is finished.

functools

The Python `functools` module is intended for higher-order functions, for example functions that act on or return other functions. Mostly you can use them as decorators, such as:

functools.cache()

Simple, lightweight, function cache as of Python 3.9, sometimes called *memoize*. It returns the same as `functools.lru_cache()` with the parameter `maxsize=None`, additionally creating a *Dictionaries* with the function arguments. Since old values never need to be deleted, this function is then also smaller and faster. Example:

```
1 >>> from functools import cache
2 >>> @cache
3 ... def factorial(n):
4 ...     return n * factorial(n-1) if n else 1
5 ...
6 >>> factorial(8)
7 40320
8 >>> factorial(10)
9 3628800
```

Line 6

Since there is no previously stored result, nine recursive calls are made.

Line 8

makes only two new calls, as the other results come from the cache.

functools.wraps()

This decorator makes the wrapper function look like the original function with its name and properties.

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwargs):
...         """Wrapper docstring"""
...         print('Call decorated function')
...         return f(*args, **kwargs)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Example docstring"""
...     print('Call example function')
...
>>> example.__name__
'example'
>>> example.__doc__
'Example docstring'
```

Without `@wraps` decorator, the name and docstring of the wrapper method would have been returned instead:

```
>>> example.__name__
'wrapper'
>>> example.__doc__
'Wrapper docstring'
```

10.2.4 Lambda functions

In Python, a lambda function is an anonymous function, that is, a function that is declared without a name. It is a small and restricted function that is no longer than one line. Like a normal function, a lambda function can have several arguments, but only one expression that is evaluated and returned.

The syntax of a lambda function is

`lambda` *`ARGUMENTS`*`:` *`EXPRESSION`*

```
>>> add = lambda x, y: x + y
>>> add(2, 3)
5
```

Note: There is no `return` statement in the lambda function. The single expression after the colon is the return value.

In the next example, a lambda function is created within a function call. However, there is no global variable to store the values of the lambda function:

```
>>> count = ['1', '123', '1000']
>>> max(count)
'123'
>>> max(count, key=lambda val: int(val))
'1000'
```

In this case, the `max()` function accepts the `key` argument, which defines how the size of each entry is to be determined. Using a lambda function that converts each string into an integer, `max` can compare the numerical values to determine the expected result.

CHAPTER

ELEVEN

MODULES

Modules are used in Python to organise larger projects. The Python standard library is divided into modules to make it more manageable. You don't have to organise your own code into modules, but if you write larger programs or code that you want to reuse, you should do so.

11.1 What is a module?

A module is a file that contains code. It defines a group of Python functions or other objects, and the name of the module is derived from the name of the file. Modules usually contain Python source code, but can also be compiled C or C++ object files. Compiled modules and Python source modules are used in the same way.

Modules not only group related Python objects together, but also help to avoid naming conflicts. You can write a module called `mymodule` for your programme that defines a function called `my_func`. In the same programme, you may also want to use another module called `othermodule`, which also defines a function called `my_func`, but does something different from your `my_func` function. Without modules, it would be impossible to use two different functions with the same name. With modules, you can refer to the functions `mymodule.my_func` and `othermodule.my_func` in your main programme. Using the module names ensures that the two `my_func` functions are not confused, as Python uses so-called namespaces. A namespace is essentially a dictionary of names for the functions, classes, modules, ETC. available there.

Modules are also used to make Python itself more manageable. Most of Python's standard functions are not integrated into the core of the language, but are provided via special modules that you can load as needed.

See also:

- Python Module Index

11.2 Creating modules

Probably the best way to learn about modules is to create your own module. To do this, we create a text file called `wc.py`, and enter the Python code below into this text file. If you use *IDLE*, select *File* → *New Window* and start typing.

It is easy to create your own modules that can be imported and used in the same way as Python's built-in library modules. The following example is a simple module with a function that prompts for a file name and determines the number of words in this file.

```
1 """wc module. Contains function: words_occur()"""
2
3
4 def words_occur():
```

(continues on next page)

(continued from previous page)

```
5       """words_occur() - count the occurrences of words in a file."""
6       # Prompt user for the name of the file to use.
7       file_name = input("Enter the name of the file: ")
8       # Open the file, read it and store its words in a list.
9       f = open(file_name, "r")
10      word_list = f.read().split()
11      f.close()
12      # Count the number of occurrences of each word in the file.
13      occurs_dict = {}
14      for word in word_list:
15          # increment the occurrences count for this word
16          occurs_dict[word] = occurs_dict.get(word, 0) + 1
17      # Print out the results.
18      print(
19          f"File {file_name} has {len(word_list)} words, "
20          f"{len(occurs_dict)} are unique:"
21      )
22      print(occurs_dict)
23
24
25  if __name__ == "__main__":
26      words_occur()
```

Lines 1 and 5

Docstrings are standard methods for documenting modules, functions, methods and classes.

Line 10

`read` returns a string containing all the characters in a file, and `split` returns a list of the words in a string using spaces.

Lines 25 to 26

With this `if`-statement you can use the programme in two ways:

- for importing in the Python shell or another Python script `__name__` is the filename:

```
>>> import wc
>>> wc.words_occur()
Enter the name of the file: README.rst
File README.rst has 350 words (187 are unique)
{'Quick': 1, ...}
```

Alternatively, you can also import `words_occur` directly:

```
>>> from wc import words_occur
>>> words_occur()
Enter the name of the file: README.rst
File README.rst has 350 words (187 are unique)
{'Quick': 1, ...}
```

You can use the interactive mode of the Python shell or *IDLE* to incrementally test a module as you create it. However, if you change your module on disk, entering the import command again will not reload it. For this purpose, you must use the `reload` function from the importlib module:

```
>>> import wc, importlib
>>> importlib.reload(wc)
<module 'wc' from '/home/veit/.local/lib/python3.8/site-packages/wc.py'>
```

- as a script it is executed with the name `__main__` and the function `words_occur()`` is called:

```
$ python3 wc.py
Enter the name of the file: README.rst
File README.rst has 350 words (187 are unique)
{'Quick': 1, ...}
```

First save this code in one of the directories of the module search path, which can be found in the list of `sys.path`. We recommend `.py` as the file name extension, as this identifies the file as Python source code.

Note: The list of directories displayed with `sys.path` depends on your system configuration. This list of directories is searched by Python in the order when an import statement is executed. The first module found that matches the import request is used. If there is no matching module in this search path, an `ImportError` is raised.

If you are using *IDLE*, you can view the search path and the modules it contains graphically by using the *File → Path Browser* window.

The variable `sys.path` is initialised with the value of the environment variable `PYTHONPATH`, if it exists. When you run a Python script, the `sys.path` variable for that script will have the directory where the script is located as the first element, so you can conveniently find out where the executing Python programme is located.

11.3 Command line arguments

In our example, if you want to pass the file name as a command line argument, for example

```
$ python3 wc.py README.rst
```

you can easily do this with the following modification of our script:

```
--- /home/docs/checkouts/readthedocs.org/user_builds/python-basics-tutorial/checkouts/24.
↪1.0/docs/modules/wc.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/python-basics-tutorial/checkouts/24.
↪1.0/docs/modules/wcargv.py
@@ -1,10 +1,12 @@
 """wc module. Contains function: words_occur()"""
+
+import sys


 def words_occur():
     """words_occur() - count the occurrences of words in a file."""
     # Prompt user for the name of the file to use.
-    file_name = input("Enter the name of the file: ")
+    file_name = sys.argv.pop()
     # Open the file, read it and store its words in a list.
     f = open(file_name, "r")
     word_list = f.read().split()
```

(continues on next page)

(continued from previous page)

```
@@ -16,8 +18,8 @@
         occurs_dict[word] = occurs_dict.get(word, 0) + 1
     # Print out the results.
     print(
-        f"File {file_name} has {len(word_list)} words, "
-        f"{len(occurs_dict)} are unique:"
+        "File %s has %d words (%d are unique)"
+        % (file_name, len(word_list), len(occurs_dict))
     )
     print(occurs_dict)
```

sys.argv
: returns a list of command line arguments passed to a Python script. `argv[0]` is the script name.

.pop
: removes the element at the given position in the list and returns it. If no index is specified, `.pop()` removes the last element in the list and returns it.

11.4 The argparse module

You can configure a script to accept command line options as well as arguments. The argparse module supports parsing of different argument types and can even generate messages. To use the `argparse` module, create an instance of `ArgumentParser`, fill it with arguments, and then read both the optional and positional arguments. The following example illustrates the use of the module:

```
--- /home/docs/checkouts/readthedocs.org/user_builds/python-basics-tutorial/checkouts/24.
↪1.0/docs/modules/wc.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/python-basics-tutorial/checkouts/24.
↪1.0/docs/modules/wcargparse.py
@@ -1,10 +1,15 @@
 """wc module. Contains function: words_occur()"""
+
+from argparse import ArgumentParser


 def words_occur():
     """words_occur() - count the occurrences of words in a file."""
+    parser = ArgumentParser()
     # Prompt user for the name of the file to use.
-    file_name = input("Enter the name of the file: ")
+    parser.add_argument("-f", "--file", dest="filename", help="read data from the file")
+    args = parser.parse_args()
+    file_name = args.filename
     # Open the file, read it and store its words in a list.
     f = open(file_name, "r")
     word_list = f.read().split()
@@ -16,8 +21,8 @@
         occurs_dict[word] = occurs_dict.get(word, 0) + 1
     # Print out the results.
     print(
```

(continues on next page)

(continued from previous page)

```
-        f"File {file_name} has {len(word_list)} words, "
-        f"{len(occurs_dict)} are unique:"
+        "File %s has %d words (%d are unique)"
+        % (file_name, len(word_list), len(occurs_dict))
     )
     print(occurs_dict)
```

This code creates an instance of `ArgumentParser` and then adds the filename argument. The `argparse` module returns a namespace object that contains the arguments as attributes. You can retrieve the values of the arguments with dot notation, in our case with `args.filename`.

You can now call the script with:

```
$ python3 wcargparse.py -f index.rst
```

In addition, a help option `-h` or `--help` is automatically generated:

```
$ python3 wcargparse.py -h
usage: wcargparse.py [-h] [-f FILENAME]

optional arguments:
  -h, --help            show this help message and exit
  -f FILENAME, --file FILENAME
                        read data from the file
```

CHAPTER

TWELVE

PROGRAMME LIBRARIES

Several *Modules* can be grouped together in a programme library. Such libraries allow you to group modules into directories and subdirectories and then import and hierarchically reference them using a `package.subpackage.module` syntax. This does not require much more than the creation of a possibly empty initialisation file for each package or subpackage.

12.1 *„Batteries included“*

In Python, a library can consist of several components, including built-in data types and constants that can be used without an import statement, such as *Numbers* and *Lists*, as well as some built-in *Functions* and *Exceptions*. The largest part of the library is an extensive collection of *Modules*. If you have Python installed, there are also several libraries available for you to use.

- *Managing data types*
- *Changing files*
- *Interacting with the operating system*
- *Use of Internet protocols*
- *Developing and debugging*

12.1.1 Managing data types

The standard library naturally contains support for the types built into Python. In addition, there are three categories in the standard library that deal with different data types: Modules for strings, datatypes and numbers.

String modules

:

Module	Description
`string`	compares with constants such as `string.digits` or `string.whitespace`
`re`	searches and replaces text with regular expressions
`struct`	interprets bytes as packed binary data
`difflib`	helps to calculate deltas, find differences between strings or sequences and create patches and diff files
`textwrap`	wraps and fills text, formats text with line breaks or spaces

See also:

- Manipulation of strings with pandas

Modules for data types

Module	Description
`datetime`, `calendar`	Time and calendar operations
`collections`	Container data types
`enum`	allows the creation of enumeration classes that bind symbolic names to constant values
`array`	Efficient arrays of numeric values
`sched`	Event scheduler
`queue`	Synchronised queue class
`copy`	Shallow and deep copy operations
`pprint`	prints Python data structures „pretty“.
`typing`	supports commenting code with hints about the types of objects, especially function parameters and return values

Modules for numbers

:

Module	Description
`numbers`	for numeric abstract base classes
`math`, `cmath`	for mathematical functions for real and complex numbers
`decimal`	for decimal fixed-point and floating-point arithmetic
`statistics`	for functions for calculating mathematical statistics
`fractions`	for rational numbers
`random`	for generating pseudo-random numbers and selections and for shuffling sequences
`itertools`	for functions that create iterators for efficient loops
`functools`	for higher-order functions and operations on callable objects
`operator`	for standard operators as functions

12.1.2 Changing files

:

Module	Description
os.path	performs common pathname manipulations
pathlib	manipulates pathnames
fileinput	iterates over multiple input files
filecmp	compares files and directories
tempfile	creates temporary files and directories
glob, fnmatch	use UNIX-like path and file name patterns
linecache	randomly accesses lines of text
shutil	performs higher level file operations
mimetypes	Assignment of file names to MIME types
pickle, shelve	enable Python object serialisation and persistence, see also *The pickle module*
csv	reads and writes CSV files
json	JSON encoder and decoder
sqlite3	provides a DB-API 2.0 interface for SQLite databases, see also *The sqlite module*
xml, xml.parsers.expat, xml.dom, xml.sax, xml.etree.ElementTree	reads and writes XML files, see also R:doc:*../save-data/xml*
html.parser, html.entities	Parsing HTML and XHTML
configparser	reads and writes Windows-like configuration files (.ini)
base64, binhex, binascii, quopri, uu	encodes/decodes files or streams
struct	reads and writes structured data to and from files
zlib, gzip, bz2, zipfile, tarfile	for working with archive files and compressions

See also:

- pandas IO tools
- Examples of serialisation formats CSV, JSON, Excel, XML/HTML, YAML, TOML und Pickle.

12.1.3 Interacting with the operating system

Module	Description
os	Various operating system interfaces
platform	Access to the identification data of the underlying platform
time	Time access and conversions
io	Tools for working with data streams
select	Waiting for I/O completion
optparse	Parser for command line options
curses	Terminal handling for character cell displays
getpass	Portable password entry
ctypes	provides C-compatible data types
threading	high-level threading interface
multiprocessing	Process-based threading interface
subprocess	Management of subprocesses

12.1.4 Use of Internet protocols

Module	descriptiong
socket, ssl	Low-level network interface and SSL wrapper for socket objects
email	Email and MIME processing package
mailbox	Manipulation of mailboxes in various formats
cgi, cgitb	Common Gateway Interface support
wsgiref	WSGI utilities and reference implementation
urllib.request, urllib.parse	Open and parse URLs
ftplib, poplib, imaplib, nntplib, smtplib, telnetlib	Clients for various Internet protocols
socketserver	Framework for network servers
http.server	HTTP server
xmlrpc.client, xmlrpc.server	XML-RPC client and server

12.1.5 Developing and debugging

Module	Description
pydoc	Documentation generator and online help system
doctest	Test examples from Python docstrings
unittest	Framework for unittests, see also *Unittest*
test.support	Utility functions for tests
trace	traces the execution of Python statements
pdb	Python debugger
logging	logging function for Python
timeit	measures the execution time of small code snippets
profile, cProfile	Python profiler
sys	System-specific parameters and functions
gc	Functions of the Python garbage collector
inspect	inspects objects live
atexit	exit handler
__future__	Future statement definitions
imp	allows access to the import internals
zipimport	imports modules from zip archives
modulefinder	finds modules used by a script

12.2 Adding more Python libraries

Although Python' *„Batteries included“* philosophy means that you can already do a lot with the default installation of Python, there will inevitably come a situation where you need functionality that is not included in Python. This section gives an overview of the options available to you.

If you are lucky, you will find the extra functionality you need in a package for your operating system – with a Windows or macOS executable installer, or a package for your Linux distribution.

This is one of the easiest ways to add a library to your Python installation, as the installer or your package manager will take care of all the details to correctly add the module to your system. In general, however, such pre-built packages are not the norm for Python software.

12.2.1 Installing Python libraries with `pip` and `venv`

If you need a third-party module that is not pre-built for your platform, you will have to turn to its source distribution. However, this brings two problems:

1. To install the source distribution, you need to find and download it.
2. Certain Python paths and permissions on your system are expected.

Python offers *pip* as a current solution to both problems. `pip` tries to find the module in the *Python Package Index* (*PyPI*), downloads it and all dependencies, and takes care of the installation. The basic syntax of `pip` is quite simple: for example, to install the popular `requests` library from the command line, all you have to do is the following:

```
$ python3.8 -m pip install requests
```

If you want to specify a particular version of a package, you can simply append the version numbers:

```
$ python3.8 -m pip install requests==2.28.1
```

or

```
$ python3.8 -m pip install requests>=2.28.0
```

Installing with the `--user` option

Often, however, you will not be able or willing to install a Python package in the main Python instance. Maybe you need a more recent version of a library, but another application still needs an older version. Or maybe you don't have sufficient administrator rights to change the system's default Python. In such cases, one possibility is to install the library with the `--user` flag: this installs the library in the home directory, where it can then only be used by you:

```
$ python3.8 -m pip install --user requests
```

See also:

- Installing Python Modules

Virtual environments

However, there is an even better option if you want to avoid installing libraries in the Python system. This option is called a *virtual environment* *virtualenv*). It is a self-contained directory structure that contains both an installation of Python and the additional packages. Because the entire Python environment is contained in the virtual environment, the libraries and modules installed there cannot collide with those in the main system or in other virtual environments, so different applications can use different versions of Python and its packages. Creating and using a virtual environment is a two-step process:

1. First we create the environment:

   ```
   $ python3 -m venv myenv
   ```

   ```
   > python -m venv myenv
   ```

   This creates the environment with Python and *pip* in a directory called `myenv`.

2. You can then activate this environment so that the next time you call `python`, it will use the Python from your new environment:

```
$ source myenv/bin/activate
```

```
> myenv\Scripts\activate.bat
```

3. You can then install Python packages for this virtual environment only:

```
(myenv) $ python -m pip install requests
```

```
(myenv) > python.exe -m pip install requests
```

4. If you want to finish your work on this project, you can deactivate the virtual environment again with

```
(myenv) $ deactivate
```

```
(myenv) > deactivate
```

See also:

- Virtual Environments and Packages

PyPI

The *Python Package Index* (*PyPI*) is the standard package index, but by no means the only repository for Python code. You can access it directly at *pypi.org* and search for packages or filter the packages by category.

12.3 Packages and programmes

12.3.1 wheels

The current standard format for distributing Python libraries and programs is the use of *wheels*. wheels are designed to make the installation of Python code more reliable and to make dependency management easier. However, the details of creating wheels are beyond the scope of this section, but full details of the requirements and process for creating wheels can be found in *Creating a distribution package*.

See also:

- Pradyun Gedam: Thoughts on the Python packaging ecosystem

12.3.2 `py2exe` and `py2app`

py2exe creates standalone Windows applications and py2app does the same for macOS. In both cases, these are single executables that can run on machines that do not have Python installed. In many ways, however, standalone executables are not ideal, as they tend to be larger and less flexible than native Python applications, but in some situations they can also be the best or only solution.

12.3.3 `freeze`

The `freeze` tool also creates an executable Python programme that runs on computers that do not have Python installed. If you want to use the `freeze` tool, you will probably need to download the Python source code.

Freezing a Python program creates C files that are then compiled and linked with a C compiler that you must have installed on your system. The application thus frozen will only run on platforms for which the C compiler used provides its executables.

See also:

- Tools/freeze

12.3.4 PyInstaller and PyOxidizer

PyInstaller and PyOxidizer bundle a Python application and all its dependencies into a single package.

12.3.5 Briefcase

Briefcase is a tool for converting a Python project into a standalone native application for Mac, Windows, Linux, iPhone/iPad and Android.

12.4 Creating a distribution package

Distribution Packages are archives that can be uploaded to a package index such as *pypi.org* and installed with *pip*.

Some of the following commands require a new version of pip, so you should make sure you have the latest version installed:

```
$ python3 -m pip install --upgrade pip
```

```
> python  -m pip install --upgrade pip
```

12.4.1 Structure

A minimal distribution package can look like this, for example:

```
dataprep
├── pyproject.toml
└── src
    └── dataprep
        ├── __init__.py
        └── loaders.py
```

12.4.2 pyproject.toml

PEP 517 and **PEP 518** brought extensible build backends, isolated builds and pyproject.toml in TOML format.

Among other things, `pyproject.toml` tells *pip* and *build* which backend tool to use to build distribution packages for your project. You can choose from a number of backends, though this tutorial uses `hatchling` by default.

A minimal yet functional `dataprep/pyproject.toml` file will then look like this, for example:

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

build-system
: defines a section describing the build system

requires
: defines a list of dependencies that must be installed for the build system to work, in our case `hatchling`.

Note: Dependency version numbers should usually be written in the requirements.txt file, not here.

---

build-backend
: identifies the entry point for the build-backend object as a dotted path. The `hatchling` backend object is available under `hatchling.build`.

Note: However, for Python packages that contain binary extensions with `Cython`, `C`, `C++`, `Fortran` or `Rust`, the *hatchling* backend is not suitable. One of the following backends should be used here:

- *setuptools*
- *scikit-build*
- *maturin*

But thatr's not all – there are other backends:

- *Flit*
- *whey*
- *poetry*
- *pybind11*
- *meson-python*

See also:

- pypackaging-native

Note: With validate-pyproject you can check your `pyproject.toml` file.

See also:

If you want to look at alternatives to `hatchling`:

- *setuptools*
- *Flit*

- poetry
- pypackaging-native

Metadata

In `pyproject.toml` you can also specify metadata for your package, such as:

```
[project]
name = "dataprep"
version = "0.1.0"
authors = [
  { name="Veit Schiele", email="veit@cusy.io" },
]
description = "A small dataprep package"
readme = "README.rst"
requires-python = ">=3.7"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: BSD License",
    "Operating System :: OS Independent",
]
dependencies = [
    "pandas",
]

[project.urls]
"Homepage" = "https://github.com/veit/dataprep"
"Bug Tracker" = "https://github.com/veit/dataprep/issues"
```

name
: is the distribution name of your package. This can be any name as long as it contains only letters, numbers, `.`, `_` and `-`. It should also not already be assigned on the *Python Package Index* (*PyPI*).

version
: is the version of the package.

  In our example, the version number has been set statically. However, there is also the possibility to specify the version dynamically, for example by a file:

  ```
  [project]
  ...
  dynamic = ["version"]
  [tool.hatch.version]
  path = "src/dataprep/__about__.py"
  ```

  The default pattern looks for a variable called `__version__` or `VERSION`, which contains the version, optionally preceded by the lower case letter `v`. The default pattern is based on **PEP 440**.

  If this is not the way you want to store versions, you can define a different regular expression with the `pattern` option.

  See also:

  - Calendar Versioning
  - ZeroVer

However, there are other version scheme plug-ins, such as hatch-semver for semantic Versioning.

With the version source plugin hatch-vcs you can also use Git tags:

```
[build-system]
requires = ["hatchling", "hatch-vcs"]
...
[tool.hatch.version]
source = "vcs"
raw-options = { local_scheme = "no-local-version" }
```

The setuptools backend also allows dynamic versioning:

```
[build-system]
requires = ["setuptools>=61.0", "setuptools-scm"]
build-backend = "setuptools.build_meta"
[project]
...
dynamic = ["version"]
[tool.setuptools.dynamic]
version = {attr = "dataprep.VERSION"}
```

See also:

- Configuring setuptools using pyproject.toml files: Dynamic Metadata

authors
is used to identify the authors of the package by name and email address.

You can also list `maintainers` in the same format.

description
is a short summary of the package, consisting of one sentence.

readme
is a path to a file containing a detailed description of the package. This is displayed on the package details page on *Python Package Index* (*PyPI*). In this case, the description is loaded from `README.rst`.

requires-python
specifies the versions of Python that are supported by your project. This will cause installers like *pip* to search through older versions of packages until they find one that has a matching Python version.

classifiers
gives the *Python Package Index* (*PyPI*) and *pip* some additional metadata about your package. In this case, the package is only compatible with Python 3, is under the BSD licence and is OS independent. You should always at least specify the versions of Python your package runs under, under which licence your package is available and on which operating systems your package runs. You can find a complete list of classifiers at https://pypi.org/classifiers/.

They also have a useful additional feature: to prevent a package from being uploaded to *PyPI*, use the special classifier `"Private :: Do Not Upload"`. *PyPI* will always reject packages whose classifier starts with `"Private ::"`.

dependencies
gibt die Abhängigkeiten für euer Paket in einem Array an.

See also:

PEP 631

urls
: lets you list any number of additional links that are displayed on the *Python Package Index* (*PyPI*). In general, this could lead to source code, documentation, task managers, ETC (et cetera).

See also:

- Declaring project metadata
- **PEP 345**

Optional dependencies

project.optional-dependencies
: allows you to specify optional dependencies for your package. You can also distinguish between different sets:

```
[project.optional-dependencies]
tests = [
    "coverage[toml]",
    "pytest>=6.0",
]
docs = [
    "furo",
    "sphinxext-opengraph",
    "sphinx-copybutton",
    "sphinx_inline_tabs"
]
```

Recursive optional dependencies are also possible with pip 21.2. For example, for `dev` you can take over all dependencies from `docs` and `test` in addition to `pre-commit`:

```
dev = [
    "dataprep[tests, docs]",
    "pre-commit"
]
```

You can install these optional dependencies, for example with:

```
$ cd /PATH/TO/YOUR/DISTRIBUTION_PACKAGE
$ python3 -m venv .
$ . bin/activate
$ python -m pip install --upgrade pip
$ python -m pip install -e '.[dev]'
```

```
> cd C:\PATH\TO\YOUR\DISTRIBUTION_PACKAGE
> python3 -m venv .
> Scripts\activate.bat
> python -m pip install --upgrade pip
> python -m pip install -e '.[dev]'
```

12.4.3 src package

When you create a new package, you shouldn't use a flat layout but the `src` layout, which is also recommended in Packaging Python Projects of the *PyPA*. A major advantage of this layout is that tests are run with the installed version of your package and not with the files in your working directory.

See also:

- Hynek Schlawack: Testing & Packaging

Note: In Python 3.11 `PYTHONSAFEPATH` can be used to ensure that the installed packages are used first.

---

dataprep
: is the directory that contains the Python files. The name should match the project name to simplify configuration and be more recognisable to those installing the package.

__init__.py
: is required to import the directory as a package. The file should be empty.

loaders.py
: is an example of a module within the package that could contain the logic (functions, classes, constants, ETC.) of your package.

12.4.4 Other files

CONTRIBUTORS.rst

See also:

- All contributors

LICENSE

You can find detailed information on this in the Licensing section.

README.rst

This file briefly tells those who are interested in the package how to use it.

See also:

- Make a README
- readme.so

If you write the document in *reStructuredText*, you can also include the contents as a detailed description in your package:

```
setup(
    ext_modules=cythonize("src/dataprep/cymean.pyx"),
```

You can also include them in your *Sphinx documentation* with `.. include:: ../../README.rst`.

CHANGELOG.rst

See also:

- Keep a Changelog
- Scriv
- changelog_manager
- github-activity
- Dinghy
- Python core-workflow blurb
- Release Drafter
- towncrier

12.4.5 Historical files or files needed for binary extensions

Before the `pyproject.toml` file introduced with **PEP 518** became the standard, `setuptools` required `setup.py`, `setup.cfg` and `MANIFEST.in`. Today, however, these files are only needed for *binary extensions* at best.

If you want to replace these files in your packages, you can do so with `hatch new --init` or ini2toml.

setup.py

A minimal and yet functional `dataprep/setup.py` can look like this, for example:

```
setup(
    ext_modules=cythonize("src/dataprep/cymean.pyx"),
```

package_dir points to the `src` directory, which can contain one or more packages. You can then use setuptools's find_packages() to find all packages in this directory.

Note: `find_packages()` without `src/` directory would package all directories with a `__init__.py` file, so also `tests/` directories.

---

setup.cfg

This file is no longer needed, at least not for packaging. `wheel` nowadays collects all required licence files automatically and `setuptools` can build universal `wheel` packages with the `options` keyword argument, for example `dataprep-0.1.0-py3-none-any.whl`.

MANIFEST.in

The file contains all files and directories that are not already covered by `packages` or `py_module`. It can look like this: `dataprep/MANIFEST.in`:

```
include LICENSE *.rst *.toml *.yml *.yaml *.ini
graft src
recursive-exclude __pycache__ *.py[cod]
```

For more instructions in `Manifest.in`, see MANIFEST.in commands.

Note: People often forget to update the `Manifest.in` file. To avoid this, you can use check-manifest in a pre-commit hook.

Note: If you want files and directories from `MANIFEST.in` to be installed as well, for example if they are runtime-relevant data, you can specify this with `include_package_data=True` in your `setup()` call.

12.4.6 Build

The next step is to create distribution packages for the package. These are archives that can be uploaded to the *Python Package Index* (*PyPI*) and installed by *pip*.

Make sure you have the latest version of `build` installed:

Now run the command in the same directory where `pyproject.toml` is located:

```
$ python -m pip install build
$ cd /PATH/TO/YOUR/DISTRIBUTION_PACKAGE
$ rm -rf build dist
$ python -m build
```

```
> python -m pip install build
> cd C:\PATH\TO\YOUR\DISTRIBUTION_PACKAGE
> rm -rf build dist
> python -m build
```

The second line ensures that a clean build is created without artefacts from previous builds. The third line should output a lot of text and create two files in the `dist` directory when finished:

```
dist
├── dataprep-0.1.0-py3-none-any.whl
└── dataprep-0.1.0.tar.gz
```

dataprep-0.1.0-py3-none-any.whl
: is a binary distribution format with the suffix `..whl`, where the filename is composed as follows:

  dataprep
  : is the normalised package name

  0.1.0
  : is the version of the distribution package

py3
: specifies the Python version and, if applicable, the C-ABI

none
: specifies whether the *Wheel* package is suitable for any OS or only specific ones

any
: any is suitable for any processor architecture, `x86_64` on the other hand only for chips with the x86 instruction set and a 64-bit architecture

dataprep-0.1.0.tar.gz
: is a *source distribution*.

See also:

The reference for the file names can be found in File name convention.

For more information on `sdist`, see Creating a Source Distribution and **PEP 376**.

12.4.7 Testing

```
$ mkdir test_env
$ cd test_env
$ python3 -m venv .
$ source bin/activate
$ python -m pip install dist/dataprep-0.1.0-py3-none-any.whl
Processing ./dist/dataprep-0.1.0-py3-none-any.whl
Collecting pandas
  Using cached pandas-1.3.4-cp39-cp39-macosx_10_9_x86_64.whl (11.6 MB)
...
Successfully installed dataprep-0.1.0 numpy-1.21.4 pandas-1.3.4 python-dateutil-2.8.2␣
↪pytz-2021.3 six-1.16.0
```

```
> mkdir test_env
> cd test_env
> python -m venv .
> Scripts\activate.bat
> python -m pip install dist/dataprep-0.1.0-py3-none-any.whl
Processing ./dist/dataprep-0.1.0-py3-none-any.whl
Collecting pandas
  Using cached pandas-1.3.4-cp39-cp39-macosx_10_9_x86_64.whl (11.6 MB)
...
Successfully installed dataprep-0.1.0 numpy-1.21.4 pandas-1.3.4 python-dateutil-2.8.2␣
↪pytz-2021.3 six-1.16.0
```

Anschließend könnt ihr die *Wheel*-Datei überprüfen mit:

```
$ mkdir test_env
$ cd !$
cd test_env
$ python3 -m venv .
$ source bin/activate
$ python -m pip install dist/dataprep-0.1.0-py3-none-any.whl
Processing ./dist/dataprep-0.1.0-py3-none-any.whl
Collecting pandas
```

(continues on next page)

(continued from previous page)

```
  Using cached pandas-1.3.4-cp39-cp39-macosx_10_9_x86_64.whl (11.6 MB)
...
Successfully installed dataprep-0.1.0 numpy-1.21.4 pandas-1.3.4 python-dateutil-2.8.2␣
↪pytz-2021.3 six-1.16.0
```

Then you can check the wheel with:

```
$ python -m pip install check-wheel-contents
$ check-wheel-contents dist/*.whl
dist/dataprep-0.1.0-py3-none-any.whl: OK
```

Alternatively, you can also install the package:

```
$ python -m pip install dist/dataprep-0.1.0-py3-none-any.whl
Processing ./dist/dataprep-0.1-py3-none-any.whl
Collecting pandas
...
Installing collected packages: numpy, pytz, six, python-dateutil, pandas, dataprep
Successfully installed dataprep-0.1 numpy-1.21.4 pandas-1.3.4 python-dateutil-2.8.2 pytz-
↪2021.3 six-1.16.0
```

You can then call Python and import your `loaders` module:

```
from dataprep import loaders
```

Note: There are still many instructions that include a step to call `setup.py`, for example `python setup.py sdist`. However, this is now considered anti-pattern by parts of the Python Packaging Authority (PyPA).

12.5 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:

PyPI packages in the Package Registry

12.5.1 Authentication

To authenticate to the GitLab Package Registry, you can use one of the following methods:

- A *personal access token* with the scope `api`.
- A *deploy token* with the scopes `read_package_registry`, `write_package_registry` or both.
- A *CI job token.*.

... with a personal access token

To authenticate yourself with a personal access token, you can add the following to the `~/.pypirc` file, for example:

```
[distutils]
index-servers=
    gitlab

[gitlab]
repository = https://ce.cusy.io/api/v4/projects/{PROJECT_ID}/packages/pypi
username = {NAME}
password = {YOUR_PERSONAL_ACCESS_TOKEN}
```

... with a deploy token

```
[distutils]
index-servers =
    gitlab

[gitlab]
repository = https://ce.cusy.io/api/v4/projects/{PROJECT_ID}/packages/pypi
username = {DEPLOY_TOKEN_USERNAME}
password = {DEPLOY_TOKEN}
```

... with a job token

```
image: python:latest

run:
  script:
    - pip install build twine
    - python -m build
    - TWINE_PASSWORD=${CI_JOB_TOKEN} TWINE_USERNAME=gitlab-ci-token python -m twine␣
↪upload --repository-url ${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/pypi dist/*
```

... for access to packages within a group

Use the *GROUP_URL* instead of the *PROJECT_ID*.

12.5.2 Publishing the distribution package

You can publish your package with the help of *twine*:

```
python3 -m twine upload --repository gitlab dist/*
```

Note: If you try to publish a package that already exists with the same name and version, you will get the error `400 Bad Request`; you will have to delete the existing package first.

12.5.3 Installing the package

You can install the latest version of your package for example with

```
pip install --index-url https://{NAME}:{PERSONAL_ACCESS_TOKEN}@ce.cusy.io/api/v4/
↪projects/{PROJECT_ID}/packages/pypi/simple --no-deps {PACKAGE_NAME}
```

… or from the group level with

```
pip install --index-url https://{NAME}:{PERSONAL_ACCESS_TOKEN}@ce.cusy.io/api/v4/groups/
↪{GROUP_ID}/-/packages/pypi/simple --no-deps {PACKAGE_NAME}
```

… or in the `requirements.txt` file with

```
--extra-index-url https://ce.cusy.io/api/v4/projects/{PROJECT_ID}/packages/pypi/simple
↪{PACKAGE_NAME}
```

12.6 Templating

With Cookiecutter, file structures can be created which simplify the creation of Python packages significantly.

See also:

- Copier

12.6.1 CookieCutter features

- Cross-platform: Windows, Mac and Linux are supported
- works with Python 3.6, 3.7, 3.8 and PyPy3
- The project templates can be created for any programming language and any markup format: Python, JavaScript, Ruby, ReST, CSS, HTML. Several languages can also be used in the same template.
- Templates can be easily adapted in the terminal:

  ```
  $ cookiecutter https://github.com/veit/cookiecutter-namespace-template
  full_name [Veit Schiele]:
  ...
  ```

- You can also use local templates:

  ```
  $ cookiecutter cookiecutter-namespace-template
  ```

- Alternatively you can also use CookieCutter with Python:

  ```
  $ bin/python
  >>> from cookiecutter.main import cookiecutter
  >>> cookiecutter('.https://github.com/veit/cookiecutter-namespace-template.git')
  full_name [Veit Schiele]:
  ...
  ```

- Directory and file names can be assigned to templates, for example:

```
{{cookiecutter.project_name}}/{{cookiecutter.namespace}}/{{cookiecutter.package_
↪name}}/{{cookiecutter.project_slug}}.py
```

- The nesting depth is unlimited
- The templating is based on Jinja
- You can simply save your template variables in a `cookiecutter.json` file, for example:

```
{
  "full_name": "Veit Schiele",
  "email": "veit@example.org",
  "github_username": "veit",
  "project_name": "vsc.example",
  "project_slug": "{{ cookiecutter.project_name.lower().replace(' ', '_').replace('-
↪', '_') }}",
  "namespace": "{{ cookiecutter.project_slug.split('.')[0] }}",
  "package_name": "{{ cookiecutter.project_slug.split('.')[1] }}",
  "project_short_description": "Python Namespace Package contains all you need to␣
↪create a Python namespace package.",
  "pypi_username": "veit",
  "use_pytest": "y",
  "command_line_interface": ["Click", "No command-line interface"],
  "version": "0.1.0",
  "create_author_file": "y",
  "license": ["MIT license", "BSD license", "ISC license", "Apache Software License␣
↪2.0", "GNU General Public License v3", "Not open source"]
}
```

- You can also save the values for several templates in `~/cookiecutterrc`:

```
default_context:
    full_name: "Veit Schiele"
    email: "veit@cusy.io"
    github_username: "veit"
cookiecutters_dir: "~/.cookiecutters/"
```

- CookieCutter templates loaded from a repository are usually stored in `~/.cookiecutters/`. Then they can be referenced directly via their directory name, e.g. with:

```
$ cookiecutter cookiecutter-namespace-package
```

12.6.2 Available templates

Python

cookiecutter-namespace-template
: Namespace template for Python packages

cookiecutter-pypackage
: Template for Python packages

cookiecutter-pytest-plugin
: Minimal CookiexrCtter template for creating Pytest plugins

cookiecutter-pylibrary
: Comprehensive template for Python packages with support for tests and Deployments (C extension support for cffi and Cython, test support for Tox, Pytest, Travis-CI, Coveralls, Codacy, and Code Climate, documentation with Sphinx, packaging checks with scrutinizer, Isort etc.

cookiecutter-python-cli
: Template for creating a Python CLI application with Click

widget-cookiecutter
: Template for creating Jupyter widgets

Ansible

cookiecutter-ansible-role-ci
: Template for Ansible roles

C

bootstrap.c
: Template for projects written in C with Autotools

cookiecutter-avr
: Template for AVR development

C++

BoilerplatePP
: cmake template with unit tests for C ++ projects

Scala

cookiecutter-scala
: Template for a Hello world example with a few libraries

cookiecutter-scala-spark
: Template for an Apache-Spark application

LaTeX/XeTeX

pandoc-talk
: Template for presentations with pandoc and XeTeX

12.6.3 Overview

A minimal CookieCutter template looks like this:

```
cookiecutter-namespace-template/
├── {{ cookiecutter.project_name }}/  <--- Project template
│   └── ...
└── cookiecutter.json                 <--- Prompts & default values
```

For jsonexample, the file `cookiecutter.json` can look like this:

```
{
  "full_name": "Veit Schiele",
  "email": "veit@example.org",
  "github_username": "veit",
  "project_name": "vsc.example",
  "project_slug": "{{ cookiecutter.project_name.lower().replace(' ', '_').replace('-', '_
↪') }}",
  "namespace": "{{ cookiecutter.project_slug.split('.')[0] }}",
  "package_name": "{{ cookiecutter.project_slug.split('.')[1] }}",
  "project_short_description": "Python Namespace Package contains all you need to create␣
↪a Python namespace package.",
  "pypi_username": "veit",
  "use_pytest": "y",
  "command_line_interface": ["Click", "No command-line interface"],
  "version": "0.1.0",
  "create_author_file": "y",
  "license": ["MIT license", "BSD license", "ISC license", "Apache Software License 2.0",
↪ "GNU General Public License v3", "Not open source"]
}
```

In addition, any number of directories and files can be created.

As a result you will get the following file structure:

```
my.package/                              <--- Value corresponding to what you enter
│                                             at the project_name prompt
│
└── ...                                   <--- Files corresponding to those in your
                                              cookiecutter's
                                              {{ cookiecutter.project_name }}/ directory
```

12.6.4 Installation

Requirements

- Python interpreter
- Path to the base directory for your Python packages

  Make sure your `bin` bindirectory is in the path. Usually this is `~/.local/` for Linux and Mac OS or `%APPDATA%\Python`. on Windows. You can find more information at site.USER_BASE.

  For bash you can enter the path in your `~/.bash_profile`:

  ```
  export PATH=$HOME/.local/bin:$PATH
  ```

  and then read the file with:

  ```
  $ source ~/.bash_profile
  ```

  Make sure the directory where CookieCutter will be installed is in your `Path` so you can go directly to it. To do this, look for *Environment Variables* on your computer and add this directory to `Path`, for example `%APPDATA%\Python\Python3x\Scripts`. Then you probably have to restart the session in order to be able to use the environment variables.

See also:

Configuring Python

Installation

```
$ python -m pip install --user cookiecutter
```

12.6.5 Advanced usage

Hooks

You can write pre- or post-generate hooks. The Jinja template variables will be integrated into the scripts, for example:

```
if 'Not open source' == '{{ cookiecutter.license }}':
    remove_file('LICENSE')
```

Variables, for example, can be validated in a pre-generate hook:

```
import re
import sys


MODULE_REGEX = r'^[_a-zA-Z][_a-zA-Z0-9]+$'

module_name = '{{ cookiecutter.module_name }}'

if not re.match(MODULE_REGEX, module_name):
    print(f'ERROR: {module_name} is not a valid Python module name!')

    # exits with status 1 to indicate failure
    sys.exit(1)
```

User config

If you use CookieCutter frequently, we recommend your own user config `~/cookiecutterrc`, e.g.:

```
default_context:
    full_name: "Veit Schiele"
    email: "veit@cusy.io"
    github_username: "veit"
cookiecutters_dir: "~/.cookiecutters/"
replay_dir: "~/.cookiecutter_replay/"
```

Replay

When calling `cookiecutter` a `json` file is created in `/.cookiecutter_replay/`, for example `~/.cookiecutter_replay/cookiecutter-namespace-template.json`:

```
{"cookiecutter": {"full_name": "Veit Schiele", "email": "veit@cusy.io", "github_username
↪": "veit", "project_name": "vsc.example", "project_slug": "vsc.example", "namespace":
↪"vsc", "package_name": "example", "project_short_description": "Python Namespace␣
↪Package contains all you need to create a Python namespace package.", "pypi_username":
↪"veit", "use_pytest": "y", "command_line_interface": "Click", "version": "0.1.0",
↪"create_author_file": "y", "license": "MIT license", "_template": "https://github.com/
↪veit/cookiecutter-namespace-template"}}
```

If you want to use this information without having to confirm them again in the command line, you can simply enter the following:

```
$ cookiecutter --replay gh:veit/cookiecutter-namespace-template
```

Alternatively, the Python API can also be used:

```
from cookiecutter.main import cookiecutter
cookiecutter('gh:'veit/cookiecutter-namespace-template, replay=True)
```

This function is helpful if you want to create a project from an updated template, for example.

Selection variables

Selection variables offer various options when creating a project. Depending on the user's choice, the template renders it differently, e.g. if in the `cookiecutter.json` file the following selection is offered:

```
{
  "license": ["MIT license", "BSD license", "ISC license", "Apache Software License 2.0",
↪ "GNU General Public License v3", "Other/Proprietary License"]
}
```

This is interpreted in `cookiecutter-namespace-template/{{cookiecutter.project_name}}/README.rst`

```
{% set is_open_source = cookiecutter.license != 'Not open source' -%}
{% if is_open_source %}
    ...
{%- endif %}

{% if is_open_source %}
    ...
{% endif %}
```

and in `cookiecutter-namespace-template/hooks/post_gen_project.py`:

```
if 'Not open source' == '{{ cookiecutter.license }}':
    remove_file('LICENSE')
```

12.6.6 cruft

One problem with cookiecutter templates is that projects based on older versions of the template become obsolete when only the template is adapted to changing requirements over time. cruft tries to simplify the transfer of changes in the *Cookiecutter-Templates*'s Git repository to projects derived from it.

The main features of cruft are:

- With `cruft check` you can quickly check if a project uses the latest version of a template. This check can also be easily integrated into CI pipelines to ensure that your projects are in sync.
- cruft also automates the update of projects from cookiecutter templates.

Installation

```
$ python3.8 -m pip install cruft
```

Create a new project

To create a new project with cruft, you can run `cruft create` *PROJECT_URL* on the command line, for example:

```
$ cruft create https://github.com/veit/cookiecutter-namespace-template
full_name [Veit Schiele]:
...
```

cruft uses *Cookiecutter* for this and the only difference in the resulting output is a `.cruft.json` file that contains the git hash of the template used as well as the specified parameters.

Tip: Certain files are rarely suitable for updating, for example test cases or `__init__` files. You can tell cruft to always skip updating these files in a project by creating the project with the arguments `--skip vsc/__init__.py --skip tests` or manually adding them to a skip section in your `.cruft.json` file:

```
{
  "template": "https://github.com/veit/cookiecutter-namespace-template",
  "commit": "521d4b2aa603aec186cd7e542295edb458ba4552",
  "skip": [
      "vsc/__init__.py",
      "tests"
  ],
  "checkout": null,
  "context": {
    "cookiecutter": {
      "full_name": "Veit Schiele",
      ...
    }
  },
  "directory": null
}
```

Updating a project

To update an existing project that was created with cruft, you can run `cruft update` in the root directory of the project. If there are updates, cruft will first ask you to review them. If you accept the changes, cruft will apply them to your project and update the `.cruft.json` file.

Checking a project

To see if a project has missed a template update, you can easily call `cruft check`. If the project is out of date, an error and exit code 1 will be returned. `cruft check` can also be added to pre-commit framework and CI pipelines to ensure projects don't become unintentionally stale.

Linking an existing project

If you have an existing project that you created in the past with Cookiecutter directly from a template, you can `cruft link` *`TEMPLATE_REPOSITORY`* to link it to the template it was created with, for example:

```
$ cruft link https://github.com/veit/cookiecutter-namespace-template
```

You can then specify the last commit of the template that updated the project, or accept the default to use the last commit.

Show diff

Over time, your project may differ greatly from the actual cookiecutter template. `cruft diff` allows you to quickly see what has changed in your local project compared to the template.

12.7 Upload package

Finally, you can deploy the package on the *Python Package Index* (*PyPI*) or another index, for example *GitLab Package Registry* or *devpi*.

For this you should register on *Test PyPI*. *Test-PyPI* is a separate instance that is intended for testing and experimentation. To set up an account there, go to https://test.pypi.org/account/register/. For more information, see Using TestPyPI.

Now you can create the `~/.pypirc` file:

```
[distutils]
index-servers=
    test

[test]
repository = https://test.pypi.org/legacy/
username = veit
```

See also:

If you'd like to automate PyPI registration, please read Careful With That PyPI.

After you are registered, you can upload your *Distribution Package* with *twine*. To do this, however, you must first install twine with:

```
$ python -m pip install --upgrade pip build twine
...
All dependencies are now up-to-date!
```

Note: Run this command before each release to ensure that all release tools are up to date.

Now you can create your *Distribution Packages* with:

```
$ cd /path/to/your/distribution_package
$ rm -rf build dist
$ python -m build
```

After installing Twine you can upload all archives in `/dist` to the Python Package Index with:

```
$ twine upload -r test -s dist/*
```

-r, --repository
: The repository to upload the package.

  In our case, the `test` section from the `~/.pypirc` file is used.

-s, --sign
: signs the files to be uploaded with GPG.

You will be asked for the password you used to register on *Test PyPI*. You should then see a similar output:

```
Uploading distributions to https://test.pypi.org/legacy/
Enter your username: veit
Enter your password:
Uploading example-0.0.1-py3-none-any.whl
100%|| 4.65k/4.65k [00:01<00:00, 2.88kB/s]
Uploading example-0.0.1.tar.gz
100%|| 4.25k/4.25k [00:01<00:00, 3.05kB/s]
```

Note: If you get an error message similar to

```
The user 'veit' isn't allowed to upload to project 'example'
```

you have to choose a unique name for your package:

1. change the `name` argument in the `setup.py` file
2. remove the `dist` directory
3. regenerate the archives

12.7.1 Check

Installation

You can use `pip` to install your package and check if it works. Create a new *virtual environment* and install your package on *Test PyPI*:

```
$ python3 -m venv test_env
$ source test_env/bin/activate
$ pip install -i https://test.pypi.org/simple/ minimal_example
```

Note: If you have used a different package name, replace it with your package name in the command above.

pip should install the package from *Test PyPI* and the output should look something like this:

```
Looking in indexes: https://test.pypi.org/simple/
Collecting minimal_example
  ...
Installing collected packages: minimal_example
Successfully installed minimal_example-0.0.1
```

You can test whether your package has been installed correctly by importing the module and referencing the `name` property that was previously ntered in `__init__.py`:

```
$ python
Python 3.7.0 (default, Aug 22 2018, 15:22:29)
...
>>> import minimal_example
>>> minimal_example.name
'minimal_example'
```

Note: The packages on *Test-PyPI* are only stored temporarily. If you want to upload a package to the real *Python Package Index* (*PyPI*), you can do so by creating an account on *pypi.org* and following the same instructions, but using `twine upload dist/*`.

README

Also check whether the `README.rst` is displayed correctly on the test PyPI page.

12.7.2 PyPI

Now register on the *Python Package Index* (*PyPI*) and make sure that two-factor authentication is activated by adding the following to the `~/.pypirc` file:

```
[distutils]
index-servers=
    pypi
    test
```

(continues on next page)

(continued from previous page)

```
[test]
repository = https://test.pypi.org/legacy/
username = veit

[pypi]
username = __token__
```

With this configuration, the name/password combination is no longer used for uploading but an upload token.

See also:

- PyPI now supports uploading via API token
- What is two factor authentication and how does it work on PyPI?

Finally, you can publish your package on PyPI:

```
$ twine upload -r pypi -s dist/*
```

Note: You cannot simply replace releases as you cannot re-upload packages with the same version number.

Note: Do not remove old versions from the Python Package Index.This only causes work for those who want to keep using that version and then have to switch to old versions on GitHub. PyPI has a yank function that you can use instead. This will ignore a particular version if it is not explicitly specified with == or ===.

See also:

- PyPI Release Checklist

12.7.3 GitHub Action

You can also create a GitHub action, which creates a package and uploads it to PyPI at every time a release is created. Such a `.github/workflows/pypi.yml` file could look like this:

```
name: Publish Python Package

 on:
   release:
     types: [created]

jobs:
  test:
    ...
  package-and-deploy:
    runs-on: ubuntu-latest
    needs: [test]
    steps:
    - name: Checkout
      uses: actions/checkout@v2
      with:
        fetch-depth: 0
```

(continues on next page)

(continued from previous page)

```
18      - name: Set up Python
19        uses: actions/setup-python@v5
20        with:
21          python-version: '3.11'
22          cache: pip
23          cache-dependency-path: '**/pyproject.toml'
24      - name: Install dependencies
25        run: |
26          python -m pip install -U pip
27          python -m pip install -U setuptools build twine wheel
28      - name: Build
29        run: |
30          python -m build
31      - name: Publish
32        env:
33          TWINE_PASSWORD: ${{ secrets.TWINE_PASSWORD }}
34          TWINE_USERNAME: ${{ secrets.TWINE_USERNAME }}
35        run: |
36          twine upload dist/*
```

Lines 3–5
This ensures that the workflow is executed every time a new GitHub release is created for the repository.

Line 12
The job waits for the `test` job to pass before it is executed.

See also:

- GitHub Actions

12.7.4 Trusted Publishers

Trusted Publishers is an alternative method for publishing packages on the *PyPI*. It is based on OpenID Connect and requires neither a password nor a token. Only the following steps are required:

1. Add a *Trusted Publishers* on PyPI

   Depending on whether you want to publish a new package or update an existing one, the process is slightly different:

   - to update an existing package, see Adding a trusted publisher to an existing PyPI project
   - to publish a new package, there is a special procedure called *Pending Publisher*; see also Creating a PyPI project with a trusted publisher

     You can also use it to reserve a package name before you publish the first version. This allows you to ensure that you can publish the package under the desired name.

     To do this, you need to create a new *Pending Publisher* in pypi.org/manage/account/publishing/ with

     – Name of the PyPI project

     – GitHub repository owner

     – Name of the workflow, for example `publish.yml`

     – Name of the environment (optional), for example `release`

2. Create an environment for the GitHub actions

   If we have specified an environment on *PyPI*, we must now also create it. This can be done in *Settings* → *Environments* for the repository. The name of our environment is `release`.

3. Configure the workflow

   To do this, we now create the `.github/workflows/publish.yml` file in our repository:

```
1  ...
2  jobs:
3    ...
4    deploy:
5      runs-on: ubuntu-latest
6      environment: release
7      permissions:
8        id-token: write
9      needs: [test]
10     steps:
11     - name: Checkout
12       ...
13     - name: Set up Python
14       ...
15     - name: Install dependencies
16       ...
17     - name: Build
18       ...
19     - name: Publish
20       uses: pypa/gh-action-pypi-publish@release/v1
```

Line 6
: This is needed because we have configured an environment in *PyPI*.

Lines 7–8
: They are required for the OpenID Connect token authentication to work.

Lines 19–20
: The package uses the github.com/pypa/gh-action-pypi-publish action to publish the package.

12.8 cibuildwheel

cibuildwheel simplifies the creation of *Python Wheels* for the different platforms and Python versions through Continuous Integration (CI) workflows. More precisely it builds manylinux, macOS 10.9+, and Windows wheels for CPython and PyPy with GitHub Actions, Azure Pipelines, Travis CI, AppVeyor, CircleCI, or GitLab CI/CD.

In addition, it bundles shared library dependencies on Linux and macOS through auditwheel and delocate.

Finally, the tests can also run against the wheels.

See also:

- Docs
- GitHub

12.8.1 GitHub Actions

To build Linux, macOS, and Windows wheels, create a `.github/workflows/build_wheels.yml` file in your GitHub repo:

```
name: Build

on:
  workflow_dispatch:
  release:
    types:
      - published
```

workflow_dispatch
: allows you to click a button in the graphical user interface to trigger a build. This is perfect for manually testing wheels before a release, as you can easily download them from *artifacts*.

  See also:

  - workflow_dispatch

release
: is executed when a tagged version is transferred.

  See also:

  - release

Now the *wheels* can be built with:

```
jobs:
  build_wheels:
    name: Build wheels on ${{ matrix.os }}
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-20.04, windows-2019, macos-11]

    steps:
      - uses: actions/checkout@v3

      - name: Build wheels
        uses: pypa/cibuildwheel@v2.15.0
```

This runs the CI workflow with the following default settings:

- `package-dir: .`
- `output-dir: wheelhouse`
- `config-file: "{package}/pyproject.toml"`

You can also extend the file to automatically upload the wheels to the *Python Package Index* (*PyPI*). For this, however, you should first create a *source distribution*, for example with:

```
  make_sdist:
    name: Make SDist
    runs-on: ubuntu-latest
    steps:
```

(continues on next page)

(continued from previous page)

```
    - uses: actions/checkout@v3
      with:
        fetch-depth: 0  # Optional, use if you use setuptools_scm
        submodules: true  # Optional, use if you have submodules

    - name: Build SDist
      run: pipx run build --sdist

    - uses: actions/upload-artifact@v3
      with:
        path: dist/*.tar.gz
```

In addition, this GitHub workflow must be set in the PyPI settings of your project:

- Creating a PyPI project with a trusted publisher
- Adding a trusted publisher to an existing PyPI project

Now you can finally upload the artefacts of both jobs to the *PyPI*:

```
  upload_all:
    needs: [build_wheels, make_sdist]
    environment: pypi
    permissions:
      id-token: write
    runs-on: ubuntu-latest
    if: github.event_name == 'release' && github.event.action == 'published'
    steps:
    - uses: actions/download-artifact@v3
      with:
        name: artifact
        path: dist

    - uses: pypa/gh-action-pypi-publish@release/v1
```

See also:

- Workflow syntax for GitHub Actions

12.8.2 GitLab CI/CD

To build Linux wheels with GitLab CI/CD, create a `.gitlab-ci.yml` file in your repository:

```
linux:
  image: python:3.8
  # make a docker daemon available for cibuildwheel to use
  services:
    - name: docker:dind
      entrypoint: ["env", "-u", "DOCKER_HOST"]
      command: ["dockerd-entrypoint.sh"]
  variables:
    DOCKER_HOST: tcp://docker:2375/
    DOCKER_DRIVER: overlay2
```

(continues on next page)

(continued from previous page)

```
    # See https://github.com/docker-library/docker/pull/166
    DOCKER_TLS_CERTDIR: ""
  script:
    - curl -sSL https://get.docker.com/ | sh
    - python -m pip install cibuildwheel==2.15.0
    - cibuildwheel --output-dir wheelhouse
  artifacts:
    paths:
      - wheelhouse/

windows:
  image: mcr.microsoft.com/windows/servercore:1809
  before_script:
    - choco install python -y --version 3.8.6
    - choco install git.install -y
    - py -m pip install cibuildwheel==2.15.0
  script:
    - py -m cibuildwheel --output-dir wheelhouse --platform windows
  artifacts:
    paths:
      - wheelhouse/
  tags:
    - windows
```

See also:

- Keyword reference for the .gitlab-ci.yml file

12.8.3 Optionen

`cibuildwheel` can be configured either via environment variables or via a configuration file such as `pyproject.toml`, for example:

```
[tool.cibuildwheel]
test-requires = "pytest"
test-command = "pytest {project}/tests"
build-verbosity = 1

# support Universal2 for Apple Silicon:
[tool.cibuildwheel.macos]
archs = ["auto", "universal2"]
test-skip = ["*universal2:arm64"]
```

See also:

- cibuildwheel: Options

12.8.4 Examples

- Coverage.py: .github/workflows/kit.yml
- matplotlib: .github/workflows/cibuildwheel.yml
- MyPy: .github/workflows/build.yml
- psutil: .github/workflows/build.yml

12.9 Binary Extensions

One of the features of the CPython interpreter is that in addition to executing Python code, it also has a rich C API available for use by other software. One of the most common uses of this C API is to create importable C extensions that allow things that are difficult to achieve in pure Python code.

12.9.1 Use Cases

The typical use cases for binary extensions can be divided into three categories:

Accelerator modules
: These modules are stand-alone and are only created to run faster than the corresponding pure Python code. Ideally, the accelerator modules always have a Python equivalent that can be used as a fallback if the accelerated version is not available on a particular system.

  The CPython standard library uses many accelerator modules.

Wrapper modules
: These modules are created to make existing C interfaces available in Python. You can either make the underlying C interfaces directly available or provide a *Pythonic* API that uses features of Python to make the API easier to use.

  The CPython standard library uses extensive wrapper modules.

Low-level system access
: These modules are created to access functions of the CPython runtime environment, the operating system or the underlying hardware. With platform-specific code, things can be achieved that would not be possible with pure Python code.

  A number of CPython standard library modules are written in C to access interpreter internals that are not available at the language level.

  A particularly noteworthy property of C extensions is that they can release the Global Interpreter Lock (GIL) of CPython for long-running operations, regardless of whether these operations are CPU or IO-bound.

Not all expansion modules fit exactly into the above categories. For example, the extension modules contained in NumPy cover all three use cases:

- They move inner loops to C for speed reasons,
- wrap external libraries in C, FORTRAN and other languages and
- use low-level system interfaces of CPython and the underlying operating system to support the concurrent execution of vectorised operations and to precisely control the memory layout of objects created.

12.9.2 Disadvantages

In the past, the main disadvantage of using binary extensions was that they made it difficult to distribute the software. Today this disadvantage due to *wheel* is hardly present. However, some disadvantages remain:

- The installation from the sources remains complicated.
- Possibly there is no suitable *wheel* for the build of the CPython interpreter or alternative interpreters such as PyPy, IronPython or Jython.
- The maintenance of the packages is more time-consuming because the maintainers not only have to be familiar with Python but also with another language and the CPython C API. In addition, the complexity increases if a Python fallback implementation is provided in addition to the binary extension.
- Finally, import mechanisms, such as direct import from ZIP files, often do not work for extension modules.

12.9.3 Alternatives

... to accelerator modules

If extensions modules are only used to make code run faster, a number of other alternatives should also be considered:

- Looks for existing optimised alternatives. The CPython standard library contains a number of optimised data structures and algorithms, especially in the builtins and the modules `collections` and `itertools`.

  Occasionally the *Python Package Index* (*PyPI*) also offers additional alternatives. Sometimes a third-party module can avoid the need to create your own accelerator module.
- For long-running applications, the JIT-compiled PyPy interpreter can be a suitable alternative to the standard CPython. The main difficulty with adopting PyPy is typically the dependence on other Binary Extensions modules. While PyPy emulates the CPython C API, modules that rely on it cause problems for the PyPy JIT, and the emulation often exposes defects in extension modules that CPython tolerates. (often with reference counting errors).
- Cython is a sophisticated static compiler that can compile most Python code into C-Extension modules. The initial compilation offers some speed increases (by bypassing the CPython interpreter level), and Cython's optional static typing functions can provide additional speed increases. For Python programmers, Cython offers a lower barrier to entry relative to other languages such as C or C ++).

  However, using Cython has the disadvantage of adding complexity to the distribution of the resulting application.
- Numba is a newer tool that uses the LLVM compiler infrastructure to selectively compile parts of a Python application to native machine code at runtime. It requires LLVM to be available on the system the code is running on. It can lead to considerable increases in speed, especially with vectorisable processes.

... to wrapper modules

The C-ABI (Application Binary Interface) is a standard for the common use of functions between several applications. One of the strengths of the CPython C-API (Application Programming Interface) is that Python users can take advantage of this functionality. However, manually wrapping modules is very tedious, so a number of other alternatives should be considered.

The approaches described below do not simplify distribution, but they can significantly reduce the maintenance effort compared to wrapper modules.

- Cython is useful not only for creating accelerator modules, but also for creating wrapper modules. Since the API still needs to be wrapped by hand, it is not a good choice when wrapping large APIs.

- cffi is the project of some PyPy developers to give developers who already know both Python and C the possibility to make their C modules available for Python applications. It makes wrapping a C module based on its header files relatively easy, even if you are not familiar with C itself.

  One of the main advantages of cffi is that it is compatible with the PyPy JIT so that CFFI wrapper modules can fully participate in the PyPy tracing JIT optimisations.

- SWIG is a wrapper interface generator that combines a variety of programming languages, including Python, with C and C ++ code.

- The `ctypes` module of the standard library is useful to get access to C interfaces, but if the header information is not available, it suffers from the fact that it only works on the C ABI level and therefore no automatic consistency check between the exported Interface and the Python code. In contrast, the alternatives above can all work on the C API and use C header files to ensure consistency.

- pythoncapi_compat can be used to write a C extension that supports multiple Python versions with a single code base. It consists of the header file `pythoncapi_compat.h` and the script `upgrade_pythoncapi.py`.

... for low-level system access

For applications that require low level system access, a binary extension is often the best option. This applies in particular to the low level access to the CPython runtime, since some operations (such as releasing the Global Interpreter Lock (GIL) are not permitted when the interpreter executes the code itself, especially when modules such as `ctypes` or `cffi` are used to Get access to the relevant C-API interfaces.

In cases where the expansion module is manipulating the underlying operating system or hardware (instead of the CPython runtime), it is sometimes better to write a normal C library (or a library in another programming language such as C++ or Rust) that provides a C-compatible ABI) and then use one of the wrapping techniques described above to make the interface available as an importable Python module.

12.9.4 Implementation

We now want to extend our `dataprep` package and integrate some C code. For this we use Cython to translate the Python code from `dataprep/src/dataprep/cymean.pyx` into optimised C code during the build process. Cython files have the suffix `pyx` and can contain both Python and C code.

However, we cannot currently use `hatchling.build` as a build backend, but instead fall back on a current version of *setuptools*:

```
dependencies = [
    "Cython",
    "pandas",
]
```

The *setuptools* use `dataprep/setup.py` to include non-Python files in a package.

```
setup(
    ext_modules=cythonize("src/dataprep/cymean.pyx"),
```

Note: With extensionlib there is a toolkit for extension modules, which does not yet contain a `hatchling` plugin.

Note: Alternatively, you could use *Meson* or *scikit-build*:

```
[build-system]
requires = ["meson-python"]
build-backend = "mesonpy"
```

```
[build-system]
requires = ["scikit-build-core"]
build-backend = "scikit_build_core.build"
```

Since Cython itself is a Python package, it can simply be added to the list of dependencies in the `dataprep/pyproject.toml` file:

```
requires = ["Cython", "setuptools>=61.0"]
```

Now you can run the build process with the `pyproject-build` command and check whether the Cython file ends up in the package as expected:

```
$ pyproject-build .
* Creating venv isolated environment...
* Installing packages in isolated environment... (cython, setuptools>=40.6.0, wheel)
* Getting dependencies for sdist...
Compiling src/dataprep/cymean.pyx because it changed.
[1/1] Cythonizing src/dataprep/cymean.pyx
...
copying src/dataprep/cymean.c -> dataprep-0.1.0/src/dataprep
copying src/dataprep/cymean.pyx -> dataprep-0.1.0/src/dataprep
...
running build_ext
building 'dataprep.cymean' extension
...
Successfully built dataprep-0.1.0.tar.gz and dataprep-0.1.0-cp39-cp39-macosx_10_9_x86_64.
↪whl
```

Finally, we can check our package with `check-wheel-contents`:

```
$ check-wheel-contents dataprep/dist/*.whl
dataprep/dist/dataprep-0.1.0-cp39-cp39-macosx_10_9_x86_64.whl: OK
```

Alternatively, you can install our `dataprep` package and use `mean`:

```
$ python -m pip install dataprep/dist/dataprep-0.1.0-cp39-cp39-macosx_10_9_x86_64.whl
$ python
```

```
>>> from dataprep.mean import mean
>>> from random import randint
>>> nums = [randint(1, 1_000) for _ in range(1_000_000)]
>>> mean(nums)
500097.867198
```

With the `random.randint` function a tlist of one million random numbers with values between 1 and 1000 was created.

See also:

The CPython Extending and Embedding guide contains an introduction to writing your own extension modules in C: Extending Python with C or C++. However, please note that this introduction only discusses the basic tools for creating

extensions that are provided as part of CPython. Third-party tools such as Cython, cffi, SWIG, and Numba offer both simpler and more sophisticated approaches to building C and C++ extensions for Python.

Python Packaging User Guide: Binary Extensions not only covers various available tools that simplify the creation of binary extensions, but also explains the various reasons why creating an extension module might be desirable.

12.9.5 Creating binary extensions

Binary extensions for Windows

Before you can create a binary extension, you have to make sure that you have a suitable compiler available. On Windows, Visual C is used to create the official CPython interpreter, and it should also be used to create compatible binary extensions:

For Python 3.5 install Visual Studio Code with Python Extension

> **Note:** Visual Studio is backwards compatible from Python 3.5, which means that any future version of Visual Studio can create Python extensions for all Python versions from version 3.5.

Building with the recommended compiler on Windows ensures that a compatible C library is used throughout the Python process.

Binary Extensions for Linux

Linux binaries must use a sufficiently old glibc to be compatible with older distributions. Distrowatch prepares in table form which versions of the distributions deliver which library:

- Red Hat Enterprise Linux
- Debian
- Ubuntu
- ...

The PYPA/Manylinux project facilitates the distribution of Binary extensions as *Wheels* for most Linux platforms. This also resulted in **PEP 513**, which defines the `manylinux1_x86_64` and `manylinux1_i686` platform tags.

Binary Extensions for Mac

Binary compatibility on macOS is determined by the target system for the minimal implementation, e.g. *10.9*, which is defined in the environment variable `MACOSX_DEPLOYMENT_TARGET`. When creating with setuptools/distutils the deployment target is specified with the flag `--plat-name`, for example `macosx-10.9-x86_64`. For more information on deployment targets for Mac OS Python distributions, see the MacPython Spinning Wheels-Wiki.

12.9.6 Deployment of binary extensions

In the following, the deployment on the *Python Package Index* (*PyPI*) or another index will be described.

Note: When deploying on Linux distributions, it should be noted that these make demands on the specific build system. Therefore, *Source Distributions (sdist)* should also be provided in addition to *Wheels*.

See also:

- Deploying Python applications
- Supporting Windows using Appveyor

12.10 Glossary

build
: `build` is a **PEP 517**-compatible Python package builder. It provides a CLI for building packages and a Python API.

  Docs | GitHub | PyPI

built distribution
bdist
: A structure of files and metadata that only needs to be moved to the correct location on the target system during installation. *wheel* is such a format, but not *distutils Source Distribution* that require a build step.

cibuildwheel
: *cibuildwheel* is a Python package that creates *wheels* for all common platforms and Python versions on most CI systems.

  Docs | GitHub | PyPI

  See also:

  multibuild

conda
: Package management tool for the Anaconda distribution from Continuum Analytics. It's specifically aimed at the scientific community, particularly Windows, where installing binary extensions is often difficult.

  Conda does not install packages from PyPI and can only install from the official Continuum repositories or from anaconda.org or local (e.g. intranet) package servers. Note, however, that Pip can be installed in conda and can work side by side to manage distributions of PyPI.

  See also:

  - Conda: Myths and Misconceptions
  - Conda build variants

devpi
: devpi is a powerful *PyPI* compatible server and PyPI proxy cache with a command line tool to enable packaging, testing and publishing activities.

  Docs | GitHub | PyPI

distribution package
: A versioned archive file that contains Python *packages*, *modules*, and other resource files used to distribute a *release*.

distutils
Python standard library package that provides support for bootstrapping *pip* into an existing Python installation or *virtual environment*.

Docs | GitHub

egg
A *built distribution* format introduced by *setuptools* that is now being replaced by *wheel*. For more information, see The Internal Structure of Python Eggs and Python Eggs.

enscons
enscons is a Python packaging tool based on SCons. It builds *pip*-compatible *source distributions* and *wheels* without using *distutils* or *setuptools*, including distributions with C extensions. enscons has a different architecture and philosophy than *distutils*, as it adds Python packaging to a general build system. enscons can help you build *sdists* and *wheels*.

GitHub | PyPI

Flit
Flit provides an easy way to build pure Python packages and modules and upload them to the *Python Package Index*. Flit can generate a configuration file to quickly set up a project, create a *source distribution* and *wheel*, and upload them to PyPI.

Flit uses *pyproject.toml* to configure a project. Flit does not rely on tools like *setuptools* to create distributions, or on *twine* to upload them to *PyPI*.

Docs | GitHub | PyPI

Hatch
Hatch is a command line tool that can be used to configure and version packages and specify dependencies. The plugin system allows you to easily extend the functionality.

Docs | GitHub | PyPI

hatchling
Build backend of *hatch*, which can also be used to publish on the *Python Package Index*.

import package
A Python module that can contain other modules or recursively other packages.

maturin
Formerly pyo3-pack, is a **PEP 621**-compatible build tool for *binary extensions* in Rust.

meson-python
Build backend that uses the Meson build system. It supports a variety of languages, including C, and is able to meet the requirements of most complex build configurations.

Docs | GitHub | PyPI

module
The basic unit of code reusability in Python, which exists in one of two types:

pure module
A module written in Python contained in a single `.py` file (and possibly associated `.pyc`- and/or `.pyo` files).

extension module
Usually a single dynamically loadable precompiled file, for example a common object file (`.so`).

multibuild
`multibuild` is a set of CI scripts for building and testing Python *wheels* for Linux, macOS and Windows.

See also:

cibuildwheel

pdm
Python package manager with **PEP 582** support. It installs and manages packages without the need to create a *virtual environment*. It also uses *pyproject.toml* to store project metadata as defined in **PEP 621**.

Docs | GitHub | PyPI

pex
Bibliothek und Werkzeug zur Erzeugung von Python EXecutable (`.pex`)-Dateien, die eigenständige Python-Umgebungen sind. .pex-Dateien sind Zip-Dateien mit `#!/usr/bin/env python` und einer speziellen `__main__.py`-Datei, die das Deployment von Python-Applikationen stark vereinfachen können.

Docs | GitHub | PyPI

pip
Popular tool for installing Python packages included in new versions of Python.

It provides the essential core functions for searching, downloading and installing packages from the *Python Package Index* and other Python package directories, and can be integrated into a variety of development workflows via a command line interface (CLI).

Docs | GitHub | PyPI

pip-tools
Set of tools that can keep your builds deterministic and still up to date with new versions of your dependencies.

Docs | GitHub | PyPI

Pipenv
Pipenv bundles *Pipfile*, *pip* and *virtualenv* into a single toolchain. It can automatically import the `requirements.txt` and also check the environment for CVEs using safety. Finally, it also facilitates the uninstallation of packages and their dependencies.

Docs | GitHub | PyPI

Pipfile
Pipfile.lock
`Pipfile` and `Pipfile.lock` are a higher-level, application-oriented alternative to *pip*'s `requirements.txt` file. The **PEP 508 Environment Markers** are also supported.

Docs | GitHub

pipx
pipx helps you avoid dependency conflicts with other packages installed on the system.

Docs | GitHub | PyPI

piwheels
Website and underlying software that fetches *source distribution* packages from *PyPI* and compiles them into binary *wheels* optimised for installation on Raspberry Pis.

Home | Docs | GitHub

poetry
An all-in-one solution for Python-only projects. It replaces *setuptools*, *venv*/*pipenv*, *pip*, *wheel* and *twine*. However, it makes some bad default assumptions for libraries and the *pyproject.toml* configuration is not standard compliant.

Docs | GitHub | PyPI

pybind11
: This is *setuptools*, but with a C++ extension and wheels generated by *cibuildwheel*.

  Docs | GitHub | PyPI

pypi.org
: pypi.org is the domain name for the *Python Package Index* (*PyPI*). In 2017 it replaced the old index domain name *pypi.python.org*. He is supported by *warehouse*.

pyproject.toml
: Tool-independent file for the specification of projects defined in **PEP 518**.

  Docs

  See also:

  - *pyproject.toml*

Python Package Index
PyPI
: *pypi.org* is the standard package index for the Python community. All Python developers can use and distribute their distributions.

Python Packaging Authority
PyPA
: The Python Packaging Authority is a working group that manages several software projects for packaging, distributing and installing Python libraries. However, the goals stated in PyPA Goals were created during discussions around **PEP 516**, **PEP 517** and **PEP 518**, which allowed competing workflows with the *pyproject.toml*-based build system that do not need to be interoperable.

readme_renderer
: `readme_renderer` is a library used to render documentation from markup languages like Markdown or reStructuredText into HTML. You can use it to check if your package descriptions are displayed correctly on *PyPI*.

  GitHub | PyPI

release
: The snapshot of a project at a specific point in time, identified by a version identifier.

  One release can result in several *Built Distributions*.

scikit-build
: Build system generator for `C`-, `C++`-, `Fortran`- and `Cython` extensions that integrates *setuptools*, *wheel* and *pip*. It uses `CMake` internally to provide better support for additional compilers, build systems, cross-compilation and finding dependencies and their associated build requirements. To speed up and parallelise the creation of large parallelisation, Ninja can also be installed. can be installed.

  Docs | GitHub | PyPI

setuptools
: setuptools are the classic build system, which is very powerful, but with a steep learning curve and high configuration effort. From version 61.0.0 setuptools also support *pyproject.toml* files.

  Docs | GitHub | PyPI

  See also:

  Packaging and distributing projects

shiv
: Command line utility for building Python zip apps as described in **PEP 441**, but additionally with all dependencies.

  Docs | GitHub | PyPI

source distribution
sdist
A distribution format (typically generated using) `python setup.py sdist`.

It provides metadata and the essential source files required for installation with a tool like *Pip* or for generating *built distributions*.

Spack
Flexible package manager that supports multiple versions, configurations, platforms and compilers. Any number of versions of packages can co-exist on the same system. Spack is designed for rapid creation of high-performance scientific applications on clusters and supercomputers.

Docs | GitHub

See also:

- Spack

trove-classifiers
trove-classifiers are classifiers used in the *Python Package Index* to systematically describe projects and make them easier to find. On the other hand, they are a package that contains a list of valid and obsolete classifiers that can be used for verification.

Docs | GitHub | PyPI

twine
Command line programme that passes programme files and metadata to a web API. This allows Python packages to be uploaded to the *Python Package Index*.

Docs | GitHub | PyPI

venv
Package that is in the Python standard library as of Python 3.3 and is intended for creating *virtual environments*.

Docs | GitHub

virtualenv
Tool that uses the `path` command line environment variable to create isolated Python *virtual environments*, similar to *venv*, but provides additional functionality for configuration, maintenance, duplication and debugging.

As of version 20.22.0, virtualenv no longer supports Python versions 2.7, 3.5 and 3.6.

Virtual environment
An isolated Python environment that allows packages to be installed for a specific application rather than system-wide.

See also:

- *Virtual environments*
- Creating Virtual Environments

Warehouse
The current code base that powers the *Python Package Index* (*PyPI*). It is hosted on *pypi.org*.

Docs | GitHub

wheel
Distribution format introduced with **PEP 427**. It is intended to replace the *Egg* format and is supported by current *pip* installations.

C extensions can be provided as platform-specific wheels for Windows, macOS and Linux on *PyPI*. This has the advantage for the users of the package that they don't have to compile during the installation.

Home | Docs | **PEP 427** | GitHub | PyPI |

See also:

- *wheels*

whey
Simple Python *wheel* builder with automation options for *trove-classifiers*.

CHAPTER

THIRTEEN

OBJECT ORIENTATION

Python offers full support for object-oriented programming OOP (object-oriented programming). The following listing is an example that could be the beginning of a simple shapes module for a drawing program.

13.1 Classes

A class in Python is actually a data type. All of Python's built-in data types are classes, and Python provides you with powerful tools to manipulate every aspect of a class's behaviour. You can define a class with the `class` statement:

```
>>> class MyClass:
...     STATEMENTS
```

`MyClass`
: Class identifiers are usually written in capital letters, that mean the first letter of each word is capitalised to emphasise the identifiers.

`STATEMENTS`
: is a list of Python statements – usually variable assignments and function definitions. However, no assignments or function definitions are required; it can just be a single `pass` statement.

After you have defined the class, you can create a new object of the class type (an instance of the class) by calling the class name as a function:

```
>>> instance = MyClass()
```

Class instances can be used as structures or data sets. However, unlike C structures or Java classes, the data fields of an instance do not have to be declared in advance. The following short example defines a class called `Square`, creates a `Square` instance, assigns a value to the edge length and then uses this value to calculate the total edge length:

```
>>> my_square = Square()
>>> my_square.length = 3
>>> print(f"The perimeter of the square is {4 * my_square.length}.")
The perimeter of the square is 12.
```

As in Java and many other languages, the fields of an instance are addressed using dot notation.

You can initialise fields of an instance automatically by including an `__init__` initialisation method in the class. This function is executed each time an instance of the class is created with this new instance as the first argument `self`. Unlike in Java and C++, Python classes can also have only one `__init__` method. In the following example, squares with an edge length of 1 are created by default:

```
>>> class Square:
...     def __init__(self):
...         self.length = 1
...
>>> my_square = Square()
>>> print(f"The perimeter of the square is {4 * my_square.length}.")
The perimeter of the square is 4.
```

Line 2

By convention, `self` is always the name of the first argument of `__init__`. `self` is set to the newly created `Square` instance when `__init__` is executed.

Line 5

Next, the code uses the class definition. You first create a `Square` instance object.

Line 6

This line takes advantage of the fact that the `length` field is already initialised.

You can also overwrite the `length` field so that the last line gives a different result than the previous `print` statement:

```
>>> my_square.length = 3
>>> print(f"The perimeter of the square is {4 * my_square.length}.")
The perimeter of the square is 12.
```

13.2 Variables

13.2.1 Instance variables

In the previous example, `length` is an instance variable of `Square` instances, which means that each instance of the class `Square` has its own copy of `length`, and the value stored in this copy may be different from the values stored in the `length` variable in other instances. In Python, you can create instance variables as needed by assigning them to the field of a class instance. If the variable does not already exist, it will be created automatically.

All uses of instance variables, both assignment and access, require explicit mention of the instance they contain, that is, `instance.variable`. A reference to a variable in itself is not a reference to an instance variable, but to a local variable in the executing method. This is different from C++ and Java, where instance variables are referenced in the same way as local function variables of the method. Python requires explicit mention of the contained instance here, and this enables a clear distinction between instance variables and local function variables.

13.2.2 Class variables

A class variable is a variable associated with a class, not an instance of a class, that can be accessed by all instances of the class. A class variable can be used to store some class-level information, such as how many instances of the class were created at a particular time. Python provides class variables, although using them requires a little more effort than in most other languages. You also need to be aware of an interaction between class and instance variables.

A class variable is created by an assignment in the class, but outside the `__init__` function. After it is created, it can be seen by all instances of the class. You can use a class variable to make a value for `pi` accessible to all instances of the `Circle` class:

```
>>> class Circle:
...     pi = 3.14159
...     def __init__(self, diameter):
...         self.diameter = diameter
...     def circumference(self):
...         return self.diameter * Circle.pi
```

Once you have entered this definition, you can query `pi` with:

```
>>> Circle.pi
3.14159
```

Note: The class variable is linked to and contained within the class that defines it. You access `Circle.pi` in this example before any `Circle` instances have been created. It is obvious that `Circle.pi` exists independently of specific instances of the `Circle` class.

You can also access a class variable from a method of a class using the class name. You do this in the definition of `Circle.circumference`, where the `circumference` function contains a special reference to `Circle.pi`:

```
>>> c = Circle(3)
>>> c.circumference()
9.424769999999999
```

However, it is unpleasant that the class name `Circle` is used in the `circumference` method to address the class variable `pi`. You can avoid this by using the special `__class__` attribute, which is available for all Python class instances. This attribute returns the class to which the instance belongs, for example:

```
>>> Circle
<class '__main__.Circle'>
>>> c.__class__
<class '__main__.Circle'>
```

The `Circle` class is internally represented by an abstract data structure, and this data structure is exactly what is obtained by the `__class__` attribute of `c`, an instance of the `Circle` class. In this example, you can retrieve the value of `Circle.pi` from `c` without explicitly referring to the name of the `Circle` class:

```
>>> c.__class__.pi
3.14159
```

You can use this code internally in the `circumference` method to get rid of the explicit reference to the `Circle` class; replace `Circle.pi` with `self.__class__.pi`.

There is a little oddity about class variables that might confuse you if you are not aware of it.

Warning: If Python searches for an instance variable and does not find an instance variable with that name, it will search for and return the value in a class variable with the same name. Only if no matching class variable can be found does Python return an error. This can be used to efficiently implement default values for instance variables; however, this also easily leads to accidentally referring to an instance variable instead of a class variable without an error being reported.

First, you can refer to the variable `c.pi`, even though `c` has no associated instance variable called `pi`. Python first tries to find such an instance variable and only when it cannot find an instance variable does it look for a class variable `pi` in `Circle`:

```
>>> c1 = Circle(1)
>>> c1.pi
3.14159
```

If you now find that your specification for `pi` has been rounded too early and you want to replace it with a more precise specification, you might be inclined to change it as follows:

```
>>> c1.pi = 3.141592653589793
>>> c1.pi
3.141592653589793
```

However, you have now only added a new instance variable `pi` to `c1`. The class variable `Circle.pi` and all other instances derived from it still have only five decimal places:

```
>>> Circle.pi
3.14159
>>> c2 = Circle(2)
>>> c1.pi
3.14159
```

13.3 Methods

A method is a function associated with a particular class. You have already seen the special `__init__` method that is called when a new instance is created. In the following example, you define another method, `circumference`, for the class `Square`; this method can be used to calculate and return the circumference for any `Square` instance. Like most custom methods, `circumference` is called with a syntax similar to accessing instance variables:

```
>>> class Square:
...     def __init__(self):
...         self.length = 1
...     def circumference(self):
...         return 4 * self.length
...
>>> s = Square()
>>> s.length = 5
>>> print(s.circumference())
20
```

The syntax for method calls consists of an instance followed by a dot followed by the method to be called on the instance. If a method is called in this way, it is a bound method call. However, a method can also be called as an unbound method by accessing it through its containing class. This practice is less practical and is almost never used because the first argument of a method called in this way must be an instance of the class in which the method is defined and is less clear:

```
>>> print(Square.circumference(s))
20
```

Like `__init__`, the `circumference` method is defined as a function within the class. The first argument of each method is the instance from which or on which it was called, by convention called `self`. In many languages, the instance is called `this` and is never explicitly passed.

Methods can be called with arguments if the method definitions accept those arguments. This version of `Square` adds

an argument to the `__init__` method so that you can create squares with a specific edge length without having to set the edge length after creating a square:

```
>>> class Square:
...     def __init__(self, length):
...         self.length = length
...     def circumference(self):
...         return 4 * self.length
```

Warning: `self.length` and `length` are not the same!

- `self.length` is the instance variable called `length`
- `length` is the local function parameter

In practice, you would probably refer to the local function parameter as `lng` or `l` to avoid confusion.

With this definition of `Square`, you can create squares with arbitrary edge lengths with a call to the `Square` class. In the following, a square with edge length `3` is created:

```
s = Square(3)
```

All of Python's standard functions – standard arguments, additional arguments, keyword arguments, ETC. – can be used with methods. You could have defined the first line of `__init__` as follows:

```
...     def __init__(self, length=1):
```

Then the call to `Square` would work with or without an additional argument; `Square()` would return a square with edge length `1` and `Square(3)` would return a square with edge length `3`.

For a method call `instance.method(arg1, arg2, ...)`, Python converts it to a normal function call by applying the following rules:

1. Search for the method name in the instance namespace. If a method has been changed or added for this instance, it is called in preference to methods in the class.
2. If the method is not found in the instance namespace, the method is searched in the class. In the previous examples, `class` is the `Square` type of the instance `s`.
3. If the method is still not found, it is searched for in a superclass, see also *Inheritance*.
4. If the method is found, it is called as a normal Python function, using instance as the first argument of the function and shifting all other arguments in the method call one space to the right. Thus `instance.method(arg1, arg2, ...)` becomes `class.method(instance, arg1, arg2, ...)`.

13.3.1 Static methods

Just like in Java, you can call static methods even if no instance of that class has been created. To create a static method, use the `@staticmethod` *decorator*:

```
"""circle module: contains the 'Circle' class"""


class Circle:
    """Circle class.
```

(continues on next page)

(continued from previous page)

```
6
7       The class variable 'circles' contains a list of all circle instances.
8
9       """
10
11      circles = []
12      pi = 3.14159
13
14      def __init__(self, diameter=1):
15          """Create a Circle instance with a given diameter and add an initialised
16          circle to the circles list."""
17          self.diameter = diameter
18          self.__class__.circles.append(self)
19
20      def circumference(self):
21          return self.diameter * self.__class__.pi
22
23      @staticmethod
24      def circumferences():
25          """Static method to sum all circle circumferences."""
26          csum = 0
27          for c in Circle.circles:
28              csum = csum + c.circumference()
29          return csum
```

Line 11
defines the class variable `circles` as an initially empty list of all `Circle` instances.

Line 14
adds initialised `Circle` instances to the `circles` list.

```
>>> import circle
>>> c1 = circle.Circle(1)
>>> c2 = circle.Circle(2)
>>> circle.Circle.circumferences()
9.424769999999999
>>> c2.diameter = 3
>>> circle.Circle.circumferences()
12.56636
```

13.3.2 Class methods

`Class methods` are similar to static methods in that they can be called before an object of the class has been instantiated. However, the class to which they belong is implicitly passed to the class methods as the first parameter:

```
23      @classmethod
24      def circumferences(cls):
25          """Class method to sum all circle circumferences."""
26          csum = 0
27          for c in cls.circles:
28              csum = csum + c.circumference()
29          return csum
```

Line 23
The `@classmethod` decorator is used before the `def` method.

Line 24
The class parameter is traditionally `cls`.

Line 27
You can use `cls` instead of `self.__class__`.

By using a class method instead of a static method, you don't have to hardcode the class name in `circumferences`.

```
>>> import circle_cm
>>> c1 = circle_cm.Circle(1)
>>> c2 = circle_cm.Circle(2)
>>> circle_cm.Circle.circumferences()
9.424769999999999
```

13.4 Inheritance

Inheritance in Python is simpler and more flexible than inheritance in compiled languages such as Java and C++ because the dynamic nature of Python does not impose as many restrictions on the language.

To see how inheritance is used in Python, let's start with the `Square` and `Circle` classes we discussed earlier and generalise them.

If we now want to use these classes in a drawing program, we need to define where on the drawing surface an instance should be located. We can do this by defining `x` and `y` coordinates for each instance:

```
>>> class Square:
...     def __init__(self, length=1, x=0, y=0):
...         self.length = length
...         self.x = x
...         self.y = y
...
>>> class Circle:
...     def __init__(self, diameter=1, x=0, y=0):
...         self.diameter = diameter
...         self.x = x
...         self.y = y
```

This approach works, but leads to a lot of repetitive code when you increase the number of shape classes, as you probably want every shape to have this positional information. This is a standard situation for using inheritance in object-oriented languages. Instead of defining the `x` and `y` variables in each shape class, you can abstract them into a general shape class and have each class that defines a particular shape inherit from that general class. In Python, this technique looks like this:

```
>>> class Form:
...     def __init__(self, x=0, y=0):
...         self.x = x
...         self.y = y
...
>>> class Square(Form):
...     def __init__(self, length=1, x=0, y=0):
```

(continues on next page)

(continued from previous page)

```
...         super().__init__(x, y)
...         self.length = length
...
>>> class Circle(Form):
...     def __init__(self, diameter=1, x=0, y=0):
...         super().__init__(x, y)
...         self.diameter = diameter
```

Lines 6 and 11
Square and Circle inherit from the Form class.

Lines 8 and 13
call the __init__ method of the Form class.

There are generally two requirements when using an inherited class in Python, both of which you can see in the code of the Circle and Square classes:

1. The first requirement is to define the inheritance hierarchy, which you do by specifying the classes that are inherited from in parentheses immediately after the name of the class, which is defined with the class keyword: Circle and Square both inherit from Form.
2. The second element is the explicit call to the __init__ method of the inherited class. This is not done automatically in Python, but mostly through the super function, more precisely through the lines super().__init__(x,y). This code calls the initialisation function of Form with the instance to be initialised and the corresponding arguments. Otherwise, the instance variables x and y would not be set for the instances of Circle and Square.

Inheritance also comes into play when you try to use a method that is not defined in the base classes but in the superclass. To see this effect, define another method in the Form class called move that moves a shape in the x and y coordinates. The definition for Form is now:

```
>>> class Form:
...     def __init__(self, x=0, y=0):
...         self.x = x
...         self.y = y
...     def move(self, delta_x, delta_y):
...         self.x = self.x + delta_x
...         self.y = self.y + delta_y
```

If you take the parameters delta_x and delta_y of the method move in the __init__ methods of Circle and Square, you can for example execute the following interactive session:

```
>>> c = Circle(3)
>>> c.move(4, 5)
>>> c.x
4
>>> c.y
5
```

The class Circle in the example does not have a move method defined directly in itself, but since it inherits from a class that implements move, all instances of Circle can use the move method. In OOP terms, one could say that all Python methods are virtual – that is if a method does not exist in the current class, the list of superclasses is searched for the method and the first one found is used.

13.5 Summary

The points made so far, are the basics of using classes and objects in Python. I will now summarise these basics in a single example:

1. First, we create a base class:

```
4  class Form:
5      """Form class: has method move"""
6
7      def __init__(self, x, y):
8          self.x = x
9          self.y = y
10
11     def move(self, deltaX, deltaY):
12         self.x = self.x + deltaX
13         self.y = self.y + deltaY
```

Line 7
The `__init__` method requires one instance (`self`) and two parameters.

Lines 8 and 9
The two instance variables `x` and `y`, which are accessed via `self`.

Line 11
The `move` method requires one instance (`self`) and two parameters.

Lines 12 and 13
Instance variables that are set in the `move` method.

2. Next, create a subclass that inherits from the base class `Form`:

```
16 class Square(Form):
17     """Square Class:inherits from Form"""
18
19     def __init__(self, length=1, x=0, y=0):
20         super().__init__(x, y)
21         self.length = length
```

Line 16
The class `Square` inherits from the class `Form`.

Line 19
`Square`'s `__init__` takes one instance (`self`) and three parameters, all with defaults.

Line 20
`Circle`'s `__init__` uses `super()` to call `Form`'s `__init__`.

3. Finally, we create another subclass that also contains a static method:

```
27 class Circle(Form):
28     """Circle Class: inherits from Form and has method area"""
29
30     circles = []
31     pi = 3.14159
32
33     def __init__(self, diameter=1, x=0, y=0):
```

(continues on next page)

(continued from previous page)

```
34          super().__init__(x, y)
35          self.diameter = diameter
36          self.__class__.circles.append(self)
37
38      def circumference(self):
39          return self.diameter * self.__class__.pi
40
41      @classmethod
42      def circumferences(cls):
43          """Class method to sum all circle circumferences."""
44          csum = 0
45          for c in cls.circles:
46              csum = csum + c.circumference()
47          return csum
```

Lines 30 and 31
`pi` and `circles` are class variables for `Circle`.

Line 33
In the `__init__` method, the instance inserts itself into the `circles` list.

Lines 38 and 39
`circumferences` is a class method and takes the class itself (`cls`) as a parameter.

Line 42
uses the parameter `cls` to access the class variable `circles`.

Now you can create some instances of the class `Circle` and analyse them. Since the `__init__` method of `Circle` has default parameters, you can create a circle without specifying any parameters:

```
>>> import form
>>> c1 = form.Circle()
>>> c1.diameter, c1.x, c1.y
(1, 0, 0)
```

If you specify parameters, they are used to set the values of the instance:

```
>>> c2 = form.Circle(2, 3, 4)
>>> c2.diameter, c2.x, c2.y
(2, 3, 4)
```

When you call the `move()` method, Python does not find a `move()` method in the `Circle` class, so it goes up the inheritance hierarchy and uses the `move()` method of `Form`:

```
>>> c2.move(5, 6)
>>> c2.diameter, c2.x, c2.y
(2, 8, 10)
```

You can also call the class method `circumferences()` of the class `Circle`, either through the class itself or through an instance:

```
>>> form.Circle.circumferences()
9.424769999999999
>>> c2.circumferences()
9.424769999999999
```

13.6 Private variables and methods

A private variable or private method is a variable that is not visible outside the methods of the class in which it is defined. Private variables and methods are useful for two reasons:

1. they increase security and reliability by selectively denying access to important parts of an object's implementation
2. they prevent naming conflicts that can arise from the use of inheritance.

A class can define a private variable and inherit it from a class that defines a private variable with the same name. Private variables make code easier to read because they explicitly state what is only used internally in a class. Everything else is the interface of the class.

Most languages that define private variables do so by using the keyword *private* or similar. The convention in Python is simpler and also makes it easier to see immediately what is private and what is not. Any method or instance variable whose name begins with a double underscore (__) but does not end is private; anything else is not.

As an example, consider the following class definition:

```
>>> class MyClass:
...     def __init__(self):
...         self.x = 1
...         self.__y = 2
...     def print_y(self):
...         print(self.__y)
...
>>> m = MyClass()
>>> print(m.x)
1
>>> print(m.__y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute '__y'
```

The `print_y` method is not private, and since it is in the `MyClass` class, it can access and output `__y`:

```
>>> m.print_y()
2
```

Note: The mechanism used to ensure privacy falsifies the name of private variables and private methods when the code is compiled into bytecode. Specifically, this means that `_classname` is prefixed to the variable name:

```
>>> dir(m)
['_MyClass__y', '__class__', ...]
```

So this is only to prevent accidental access.

13.7 @property decorator

In Python, you can access instance variables directly, without additional getter and setter methods that are often used in Java and other object-oriented languages. This makes writing Python classes cleaner and easier, but in some situations using getter and setter methods can also be useful. Let's say you need a value before setting it in an instance variable, or you just want to find out the value of an attribute. In both cases, getter and setter methods would do the job, but at the cost of losing easy access to instance variables in Python.

The answer is to use a *property*. This combines the ability to pass access to an instance variable via methods such as getters and setters with simple access to instance variables via dot notation. To create a *property*, the `property` decorator is used with a method that has the name of the property:

```
    @property
    def length(self):
        return self.__length
```

Without a setter, however, the *property* `length` is read-only:

```
>>> s1 = form.Square()
>>> s1.length = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

To change this, you need to add a setter:

```
    @length.setter
    def length(self, new_length):
        self.__length = new_length
```

Now you can use the dot notation to both get and set the property `length`. Note that the name of the method remains the same, but the decorator changes to the *property* name, in our case to `length.setter`:

```
>>> s1 = form.Square()
>>> s1.length = 2
>>> s1.circumference()
8
```

A big advantage of Python's ability to add properties is that you can work with plain old instance variables at the beginning of development and then seamlessly switch to *property* variables whenever and wherever you need to, without changing the client code. The access is still the same, using dot notation.

13.8 Namespaces

If you are in the method of a class, you have direct access

1. to the **local namespace** with the parameters and variables declared in this method,
2. the **global namespace** with functions and variables declared at module level, and
3. the **built-in namespace** with the built-in functions and built-in exceptions.

These three namespaces are searched in this order.

To explain the different namespaces in more detail in our example, we have extended our existing module to make it clear what can be accessed within a method: `form_ns.py`.

You can get an overview of the methods that are available in a namespace with

```
    def namespaces(self):
        print("Global namespace:", list(globals().keys()))
        print("Superclass namespace:", dir(Form))
        print("Class namespace:", dir(Circle))
        print("Instance namespace:", dir(self))
        print("Local namespace:", list(locals().keys()))
```

```
>>> import form_ns
>>> c1 = form_ns.Circle()
>>> c1.namespaces()
Global namespace: ['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__
↪file__', '__cached__', '__builtins__', 'Form', 'Square', 'Circle']
Superclass namespace: ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__
↪eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '_
↪_init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
↪ '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__
↪', '__weakref__', 'move']
Class namespace: ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
↪ '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_
↪subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__
↪reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
↪'__weakref__', 'circles', 'circumference', 'circumferences', 'diameter', 'instance_
↪variables', 'move', 'namespaces', 'pi']
Instance namespace: ['_Circle__diameter', '__class__', '__delattr__', '__dict__', '__dir_
↪_', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash_
↪_', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__
↪new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str_
↪_', '__subclasshook__', '__weakref__', 'circles', 'circumference', 'circumferences',
↪'diameter', 'instance_variables', 'move', 'namespaces', 'pi', 'x', 'y']
Local namespace: ['self']
```

Via the `self` variable you also have access to

1. the **namespace of the instance** with
    - instance variables
    - private instance variables and
    - instance variables of the superclass,
2. the **namespace of the class** with
    - methods,
    - class variables,
    - private methods and
    - private class variables and
3. the **namespace of the superclass** with
    - methods of the superclass and
    - class variables of the superclass.

These three namespaces are also searched in this order.

You can now analyse the namespace of the instance with the method `instance_variables`, for example:

```
    def instance_variables(self):
        print(
            "Instance variables self.__diameter, self.x, self.y:",
            self.__diameter,
            self.x,
            self.y,
        )
```

```
>>> import form_ns
>>> c1 = form_ns.Circle()
>>> c1.instance_variables()
Instance variables self.__diameter, self.x, self.y: 1 0 0
```

Note: While you can access the `move` method of the superclass `form` with `self`, private instance variables, private methods and private class variables of the superclass are not accessible in this way.

If you only want to change instances of a certain class, you can do this with the `garbage collector`, for example:

```
>>> import forms
>>> c1 = forms.Circle()
>>> c2 = forms.Circle(2, 3, 4)
>>> s1 = forms.Square(5, 6, 7)
>>> import gc
>>> for obj in gc.get_objects():
...     if isinstance(obj, forms.Circle):
...         obj.move(3, 0)
...
>>> c1.x, c1.y
(3, 0)
>>> c2.x, c2.y
(6, 4)
>>> s1.x, s1.y
(6, 7)
```

13.9 Data types as objects

By now you have learned the basic Python *data types* and know how to create your own data types using *Classes*. Note that Python is dynamically typed, which means that the types are determined at runtime, not compile time. This is one of the reasons why Python is so easy to use. You can simply try the following:

```
>>> type(3)
<class 'int'>
>>> type('Hello')
<class 'str'>
>>> type(['Hello', 'Pythonistas'])
<class 'list'>
```

In these examples you can see the built-in `type` function in Python. It can be applied to any Python object and returns the type of the object. In this example, the function tells you that `3` is an `int` (integer), that `'Hello'` is a `str` (string)

and that `['Hello', 'Pythonistas']` is a `list`.

Of greater interest, however, may be the fact that Python returns objects in response to calls to `type`; `<<class 'int'>`, `<<class 'str'>` and `<<class 'list'>` are the screen representations of the returned objects. So you can compare these Python objects with each other:

```
>>> type('Hello') == type('Pythonistas!')
True
>>> type('Hello') == type('Pythonistas!') == type(['Hello', 'Pythonistas'])
False
```

With this technique you can, among other things, perform a type check in your function and method definitions. However, the most common question about the types of objects is whether a particular object is an instance of a class. An example with a simple inheritance hierarchy makes this clearer:

1. First, we define two classes with an inheritance hierarchy:

   ```
   >>> class Form:
   ...     pass
   ...
   >>> class Square(Form):
   ...     pass
   ...
   >>> class Circle(Form):
   ...     pass
   ```

2. Now you can create an instance `c1` of the class `Circle`:

   ```
   >>> c1 = Circle()
   ```

3. As expected, the `type` function on `c1` outputs that `c1` is an instance of the class `Circle` defined in your current `__main__` namespace:

   ```
   >>> type(c1)
   <class '__main__.Circle'>
   ```

4. You can also get exactly the same information by accessing the `__class__` attribute of the instance:

   ```
   >>> c1.__class__
   <class '__main__.Circle'>
   ```

5. You can also explicitly check whether the two class objects are identical:

   ```
   >>> c1.__class__ == Circle
   True
   ```

6. However, two built-in functions provide a more user-friendly way of obtaining most of the information normally required:

   `isinstance()`
   : determines whether, for example, a class passed to a function or method is of the expected type.

   `issubclass()`
   : determines whether one class is the subclass of another.

```
>>> issubclass(Circle, Form)
True
>>> issubclass(Square, Form)
True
>>> isinstance(c1, Form)
True
>>> isinstance(c1, Square)
False
>>> isinstance(c1, Circle)
True
>>> issubclass(c1.__class__, Form)
True
>>> issubclass(c1.__class__, Square)
False
>>> issubclass(c1.__class__, Circle)
True
```

13.9.1 Duck typing

The use of `type`, `isinstance()` and `issubclass()` makes it fairly easy to correctly determine the inheritance hierarchy of an object or class. However, Python also has a feature that makes using objects even easier: duck typing – *„If it walks like a duck and it quacks like a duck, then it must be a duck“*. This refers to Python's way of determining whether an object is the required type for an operation, focusing on the interface of an object. In short, in Python you don't have to worry about type-checking function or method arguments and the like, but instead rely on readable and documented code in conjunction with tests to ensure that an object „quacks like a duck when needed.“

Duck typing can increase the flexibility of well-written code and, in combination with advanced object-oriented functions, gives you the ability to create classes and objects that cover almost any situation. Such special methods are attributes of a class with special meaning for Python. While they are defined as methods, they are not intended to be called directly; instead, they are called automatically by Python in response to a request to an object of that class.

One of the simplest examples of a special method is `object.__str__()`. When defined in a class, the `__str__` method attribute is called whenever an instance of that class is used and Python requires a user-readable string representation of that instance. To see this attribute in action, we again use our `Form` class with the standard `__init__` method to initialise instances of the class, but also a `__str__` method to return strings representing instances in a readable format:

```
>>> class Form:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def __str__(self):
...         return "Position: x={0}, y={1}".format (self.x, self.y)
...
>>> f = Form(2,3)
>>> print(f)
Position: x=2, y=3
```

Even though our special `__str__` method attribute was not explicitly called by our code, it could still be used by Python because Python knows that the `__str__` attribute, if present, defines a method for converting objects into user-readable strings. And this is exactly what distinguishes the special method attributes. For example, it is often a good idea to define the `__str__` attribute for a class so that you can call `print(instance)` in debugging code and get an informative statement about your object.

Conversely, however, it may be surprising that an object type reacts differently to special method attributes. Therefore, I usually use special method attributes only in one of the following two cases:

- in a commonly used class, usually for sequences, that behaves similarly to a Python built-in type, and which is made more useful by special method attributes.
- in a class that behaves almost identically to a built-in class, for example lists implemented as balanced trees to speed up insertion, I can define the special method attributes.

CHAPTER

FOURTEEN

SAVE AND ACCESS DATA

To store data persistently, a process called serialisation or marshalling can be used. In it, data structures are converted into a linear form and stored. The reverse process is then called deserialisation or unmarshalling. Python offers several modules in the standard library that you can be used to serialise and deserialise objects:

the marshal module
: is mainly used internally by Python and should not be used to store data in a backwards compatible way.

the *pickle* module
: if you don't need a readable format or interoperability.

the json module
: you can use to exchange data for different languages in a readable form.

the *xml* module
: you can also use to exchange data in different languages in a readable form.

14.1 The Python Database API

The Python Database API (Application Programming Interface) defines a standard interface for Python database access modules. It's defined in **PEP 249** and widely used, for example by *sqlite*, *psycopg*, and mysql-python.

14.2 SQLAlchemy

SQLAlchemy is a widely used database toolkit. It provides not only an ORM (Object Relational Mapper) but also a generalised API for writing database-agnostic code without SQL. Alembic is based on SQLAlchemy and serves as a database migration tool.

14.3 NoSQL databases

There is data that is difficult to transfer into a relational data model. At the least then you should take a look at NoSQL databases.

14.3.1 File system

To work with files, you often have to interact with the file system and the different conventions depending on the operating system. For this I show you os and especially os.path.

Paths and path names

All operating systems refer to files with strings called pathnames. Python provides a number of functions to help you solve some problems. The semantics of pathnames are very similar on all operating systems because the file system is usually modelled as a tree structure, with a hard disk representing the root and folders, subfolders, ETC. representing the branches and subbranches; this means that most operating systems refer to a particular file in a very similar way.

However, different operating systems have different conventions for path names. The character used to separate consecutive file or directory names in a Linux/macOS pathname is /, while in a Windows pathname it is \. Also, the Linux file system has a single root directory referred to by a / character as the first character in the path name, while the Windows file system has a separate root directory for each drive, referred to as {C:}, and so on. Because of these differences, files have different path names on different operating systems. A file named *C:\DATA\MYFILE* on Windows could be */DATA/MYFILE* on Linux and macOS. Python provides functions and constants that allow you to perform common pathname manipulations without having to worry about such syntactical details. With a little care, you can write your Python programs to run correctly regardless of the underlying file system.

Absolute and relative paths

These operating systems allow two types of path names:

Absolute path names
uniquely indicate the exact position of a file in the file system by listing the entire path to that file, starting with the root directory of the file system.

Two absolute Windows path names are given here as examples:

```
C:\Program Files\Python 3.9\
D:\backup\2022\06\
```

And here are two absolute Linux path names and one absolute macOS path name:

```
/bin/python3
/cdrom/backup/2022/06/
/Applications/Python\ 3.10/
```

Relative pathnames
indicate the position of a file relative to another point in the file system, and this other point is not indicated in the relative path name itself.

As example, a Windows relative pathname is given here:

```
save-data\filesystem.rst
```

... and here a relative Linux/macOS pathname:

```
save-data/filesystem.rst
```

Relative paths therefore require a context in which they are anchored. This context is usually provided in one of two ways:

- The relative path is appended to an existing absolute path, creating a new absolute path. If you have a Windows relative path *Start Menu\Programs\Python 3.8* and an absolute path *C:\Users\Veit*, then by appending the relative path a new absolute path: `C:\Users\Veit\Start Menu\Programs\Python 3.8` can be created. If you append the same relative path to another absolute path (for example to `C:\Users\Tim`, you will get a new path referring to another `HOME` directory (*Tim*).
- Relative paths can also be given a context by implicitly referring to the current working directory, that is the directory in which a Python programme is located at the time it is executed. Python commands can implicitly refer back to the current working directory if a relative path is passed to them as an argument. For example, if you use the `os.listdir('RELATIVE/PATH')` command with a relative path argument, the anchor for that relative path is the current working directory, and the result of the command is a list of the filenames in the directory whose path is formed by appending the current working directory to the relative path argument.

  The directory in which a Python file is located is called the current working directory. This directory will usually be different from the directory where the Python interpreter is located. To illustrate this, let's start Python and use the command `os.getcwd()` to find out the current working directory of Python:

```
>>> import os
>>> os.getcwd()
'/home/veit'
```

Note: `os.getcwd()` is used as a function call without arguments to make it clear that the returned value is not a constant, but changes when you change the value of the current working directory. In the example above, the result is the home directory on one of my Linux machines. On Windows machines, additional backslashes would be added to the path: `C:\\Users\\Veit`, because Windows uses the backslash \ as a path separator, but it has a different meaning in strings.

To display the contents of the current directory, you can enter the following:

```
>>> os.listdir(os.curdir)
['.gnupg', '.bashrc', '.local', '.bash_history', '.ssh', '.bash_logout', '.
↪profile', '.idlerc', '.viminfo', '.config', 'Downloads', 'Documents', '.
↪python_history']
```

However, you can also change to another directory and then have the current working directory displayed:

```
>>> os.chdir('Downloads')
>>> os.getcwd()
'/home/veit/Downloads'
```

Change path names

Python provides some ways to change pathnames with the os.path submodule without having to explicitly use an operating system-specific syntax.

`os.path.join()`
: constructs path names for different operating systems, for example under Windows:

```
>>> import os
>>> print(os.path.join('save-data', 'filesystem.rst'))
save-data\filesystem.rst
```

Here, the arguments are interpreted as a series of directory or file names to be joined into a single string that is understood by the underlying operating system as a relative path. Under Windows, this means that the names of the path components are connected with backslashes (\).

If you do the same under Linux/macOS, on the other hand, you will get / as the separator:

```
>>> import os
>>> print(os.path.join('save-data', 'filesystem.rst'))
save-data/filesystem.rst
```

You can therefore use this method to create file paths independently of the operating system on which your programme is running.

The arguments do not necessarily have to be individual directory or file names either; they can also be sub-paths that are then joined together to form a longer path name. The following example illustrates this under Windows, where either slashes (/) or double backslashes (\\) can be used in the strings:

```
>>> import os
>>> print(os.path.join('python-basics-tutorial-de\\docs', 'save-data\\filesystem.rst
↪'))
python-basics-tutorial-de\docs\save-data\filesystem.rst
```

`os.path.split()`
: returns a tuple with two elements that separates the base name of a path from the rest of the path, for example under macOS:

```
>>> import os
>>> print(os.path.split(os.getcwd()))
('/home/veit/python-basics-tutorial-de', 'docs')
```

`os.path.basename()`
: returns only the base name of the path:

```
>>> import os
>>> print(os.path.basename(os.getcwd()))
docs
```

`os.path.dirname()`
: returns the path up to the base name:

```
>>> import os
>>> print(os.path.dirname(os.getcwd()))
/home/veit/python-basics-tutorial-de
```

`os.path.splitext()`
: outputs the dotted extension notation used by most file systems to indicate the file type:

```
>>> import os
>>> print(os.path.splitext('filesystem.rst'))
('filesystem', '.rst')
```

The last element of the returned tuple contains the dotted extension of the specified file.

`os.path.commonpath()`
: is a more specialised function to manipulate path names. It finds the common path for a group of paths and is thus good for finding the lowest level directory that contains each file in a group of files:

```
>>> import os
>>> print(os.path.commonpath(['save-data/filesystem.rst', 'save-data/index.rst']))
save-data
```

`os.path.expandvars()`
: expands environment variables in paths:

```
>>> os.path.expandvars('$HOME/python-basics-tutorial')
'/home/veit/python-basics-tutorial'
```

Useful constants and functions

`os.name`
: returns the name of the Python module that was imported to handle the operating system specific details, for example:

```
>>> import os
>>> os.name
'nt'
```

Note: Most versions of Windows, with the exception of Windows CE, are identified as `nt`.

On macOS and Linux, the answer is `posix`. Depending on the platform, you can perform special operations with this answer:

```
>>> import os
>>> if os.name == 'posix':
...     root_dir = '/'
... elif os.name == 'nt':
...     root_dir = 'C:\\'
... else:
...     print('The operating system was not recognised!')
```

Getting information about files

File paths show files and directories on your hard disk. To find out more about them, there are several Python functions, including

`os.path.exists()`
: returns `True` if its argument is a path that matches a path that exists in the filesystem.

`os.path.isfile()`
: returns `True` if and only if the given path points to a file, and returns `False` otherwise, including the possibility that the path argument points to nothing in the filesystem.

`os.path.isdir()`
: returns `True` if and only if its path argument points to a directory; otherwise it returns `False`.

Other similar functions provide more specific queries:

`os.path.islink()`
: returns `True` if a path specifies a file that is a link. However, Windows link files with the extension `.lnk` are not real links in this sense and return `False`. Links created only with `mklink()` also return `True`.

`os.path.ismount()`
: returns True on `possix` filesystems if the path is a mount point.

`os.path.samefile()`
: returns `True` if the two path arguments point to the same file.

`os.path.isabs()`
: returns `True` if its argument is an absolute path; otherwise returns `False`.

`os.path.getsize()`
: returns the size of the file or directory.

`os.path.getmtime()`
: specifies the modification date of the file or directory.

`os.path.getatime()`
: gives the last access time for a file or directory.

Other file system operations

Python has other very useful commands in the `os` module: Below I describe only some cross-operating system operations, but more specific file system functions are also provided.

`os.rename()`
: names or moves a file or directory, for example

```
>>> os.rename('filesystem.rst', 'save-data/filesystem.rst')
```

`os.remove()`
: deletes files, for example

```
>>> os.remove('filesystem.rst')
```

`os.rmdir()`
: deletes an empty directory. To remove non-empty directories, use `shutil.rmtree()`; this function recursively removes all files in a directory tree.

`os.makedirs()`
: creates a directory with all necessary intermediate directories, for example

```
>>> os.makedirs('save-data/filesystem')
```

Processing all files in a directory

A useful function for recursively walking through directory structures is `os.walk()`. You can use it to walk an entire directory tree and, for each directory, return the path of that directory, a list of its subdirectories and a list of its files. It can have three optional arguments: `os.walk(directory, topdown=True, onerror=None, followlinks=False)`.

`directory`
: is the path of the starting directory

topdown
: on `True` or not present, processes the files in each directory before the subdirectories, resulting in a listing that starts at the top and goes down;

  on `False`, the subdirectories of each directory are processed first, resulting in a traversal of the tree from bottom to top.

onerror
: can be set to a function to handle errors resulting from calls to `os.listdir()`, which are ignored by default. Usually symbolic links are not followed unless you specify the parameter `follow-links=True`.

```
>>> import os
>>> for root, dirs, files in os.walk(os.curdir):
...     print("{0} has {1} files".format(root, len(files)))
...     if ".ipynb_checkpoints" in dirs:
...         dirs.remove(".ipynb_checkpoints")
...
. has 13 files
./control-flows has 13 files
./save-data has 30 files
./test has 15 files
./test/coverage has 3 files
...
```

Line 4
: checks for a directory called `.ipynb_checkpoints`.

Line 5
: removes `.ipynb_checkpoints` from the directory list.

`shutil.copytree()` recursively makes copies of all files in a directory and all its subdirectories, preserving information about access and modification times. `shutil` also has the already mentioned `shutil.rmtree()` function for removing a directory and all its subdirectories, and several functions for making copies of individual files.

14.3.2 The `pickle` module

Python can write any data structure to a file, read that data structure back out of the file, and recreate it with just a few commands. This capability can be very useful because it can save you many pages of code that does nothing but write the state of a programme to a file and read that state back in.

Python provides this capability via the pickle module. Pickle is powerful, but simple to use. Suppose that the entire state of a programme is stored in three variables: `a`, `b` and `c`. You can store this state in a file called `data.pickle` as follows:

1. Importing the `pickle` module

   ```
   >>> import pickle
   ```

2. Define different data

   ```
   >>> a = [1, 2.0, 3+4j]
   >>> b = ("character string", b"byte string")
   >>> c = {None, True, False}
   ```

3. Writing the data

```
>>> with open('data.pickle', 'wb') as f:
...     pickle.dump(a, f)
...     pickle.dump(b, f)
...     pickle.dump(c, f)
```

It does not matter what was stored in the variables. The content can be as simple as numbers or as complex as a list of dictionaries containing instances of user-defined classes. `pickle.dump()` saves everything.

The pickle module can store almost anything in this way. It can handle *Numbers*, *Lists*, *Tuples*, *Dictionaries*, *Strings* and pretty much anything made up of these object types, including all class instances. It also handles shared objects, cyclic references and other complex storage structures correctly by storing shared objects only once and restoring them as shared objects, not as identical copies.

4. Loading pickled data:

   This data can be read in again during a later programme run with `pickle.load()`:

```
>>> with open('data.pickle', 'rb') as f:
...     first = pickle.load(f)
...     second = pickle.load(f)
...     third = pickle.load(f)
```

5. Output the pickled data:

```
>>> print(first, second, third)
[1, 2.0, (3+4j)] ('character string', b'byte string') {False, None, True}
```

However, in most cases you will not want to restore all your data in the order it was saved. A simple and effective way to restore only the data of interest is to write a save function that stores all the data you want to save in a dictionary and then use Pickle to save the dictionary. You can then use a complementary restore function to read the dictionary back in and assign the values in the dictionary to the appropriate programme variables. If you use this approach with the previous example, you will get the following code:

```
>>> def save():
...     # Serialise Python objects
...     data = {'a': a, 'b': b, 'c': c}
...     # File with pickles
...     with open('data.pickle', 'wb') as f:
...         pickle.dump(data, f)
```

You can then output the data from `c` with

```
>>> with open('data.pickle', 'rb') as f:
...     saved_data = pickle.load(f)
...     print(saved_data['c'])
...
{False, None, True}
```

In addition to `pickle.dump()` and `pickle.load()`, there are also the functions `pickle.dumps()` and `pickle.loads()`. The appended s indicates that these functions process strings.

Warning: Although using a pickled object in the previous scenario can make sense, you should also be aware of the disadvantages of pickling:

- Pickling is neither particularly fast nor space-saving as a means of serialisation. Even using json to store serialised objects is faster and results in smaller files on disk.
- Pickling is not secure, and loading a pickle with malicious content can lead to the execution of arbitrary code on your machine. Therefore, you should avoid pickling if there is a possibility that the pickle file is accessible to someone who could modify it.
- Pickle versions are not always backwards compatible.

See also:

- Python-Module-Dokumentation
- Using Pickle

14.3.3 The `xml` module

The XML module comes with Python. In the following section we will focus on the two sub-modules minidom and ElementTree.

Working with `minidom`

In the following example we analyse `books.xml`:

```
<?xml version="1.0"?>
<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>
```

(continues on next page)

(continued from previous page)

```
    <date>2020-04-13</date>
  </book>
</catalog>
```

1. To do this, we first import the `minidom` module and give it the same name so that it can be referenced more easily:

```
import xml.dom.minidom as minidom
```

2. Then we define the method `getTitles` and capture the desired XML tags with the method `getElementsByTagName`:

```
def getTitles(xml):
    """
    Print all titles found in books.xml
    """
    doc = minidom.parse(xml)
    node = doc.documentElement
    books = doc.getElementsByTagName("book")
```

3. Then we create an empty list called `titles`, which is filled with the title objects:

```
    titles = []
    for book in books:
        titleObj = book.getElementsByTagName("title")[0]
        titles.append(titleObj)
```

4. Now the title is output in nested `for`-loops:

```
    for title in titles:
        nodes = title.childNodes
        for node in nodes:
            if node.nodeType == node.TEXT_NODE:
                print(node.data)
```

5. Finally, we set the `__name__` variable like `__main__` so that the module can be executed like the main program. Then we apply our `getTitles` method to our `books.xml` file:

```
if __name__ == "__main__":
    document = "books.xml"
    getTitles(document)
```

Parsing with ElementTree

1. Importing `cElementTree`:

```
import xml.etree.cElementTree as ET
```

Note: `cElementTree` written in C and is considerably faster than `ElementTree`.

---

2. Then we define the method `parseXML` and the `root` element:

```
def parseXML(xml_file):
    """
    Parse XML with ElementTree
    """
    tree = ET.ElementTree(file=xml_file)
    print(tree.getroot())
    root = tree.getroot()
    print(f"tag={root.tag}, attrib={root.attrib}")
```

```
<Element 'catalog' at 0x10b009620>
tag=catalog, attrib={}
```

3. Output the XML child elements of book:

```
    for child in root:
        print(child.tag, child.attrib)
        if child.tag == "book":
            for step_child in child:
                print(step_child.tag)
```

```
book {'id': '1'}
title
language
author
license
date
book {'id': '2'}
...
```

4. Output the contents of the child elements with `iter`:

```
    print("-" * 20)
    print("Iterating using iter")
    print("-" * 20)
    books = root.iter()
    for book in books:
        book_children = book.iter()
        for book_child in book_children:
            print(f"{book_child.tag}={book_child.text}")
```

```
--------------------
Iterating using iter
--------------------
catalog=
book=
title=Python basics
language=en
author=Veit Schiele
license=BSD-3-Clause
date=2021-10-28
book=
title=Jupyter Tutorial
...
```

14.3.4 The sqlite module

The most important features of SQLite are:

- self-contained
- serverless
- config free
- transactional

SQLite is used to save data locally, e.g. in mobile phones (Android, iOS) and in browsers (Firefox, Safari, Chrome), and many other applications.

See also:

- sqlite home
- sqlite3 — DB-API 2.0 interface for SQLite databases
- W3Schools SQL tutorial

14.3.5 Create a database

1. Import the sqlite module

```
import sqlite3
```

2. Create a database

```
conn = sqlite3.connect("library.db")

cursor = conn.cursor()
```

3. Create a table

```
cursor.execute(
    """CREATE TABLE books
                (title text, language text, author text, license text,
                        release_date text)
            """
)
```

14.3.6 Create data

1. Insert a record into the database:

```
cursor.execute(
    """INSERT INTO books
                VALUES ('Python basics', 'en', 'Veit Schiele', 'BSD',
                        '2021-10-28')"""
```

2. Save data to database:

```
conn.commit()
```

3. Insert multiple records using the more secure ? method where the number of ? should correspond to the number of columns:

```
new_books = [
    ("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"),
]
cursor.executemany("INSERT INTO books VALUES (?,?,?,?,?)", new_books)
conn.commit()
```

14.3.7 Create data from csv

1. Import the sqlite and csv modules

```
import csv
import sqlite3
```

2. Point to the Library Database

```
conn = sqlite3.connect("library.db")
cursor = conn.cursor()
```

3. Read the csv file and insert the records into the database:

```
with open("books.csv", encoding="utf-8") as f:
    reader = csv.reader(f, delimiter=",")
    cursor.executemany("INSERT INTO books VALUES (?,?,?,?,?)", reader)
```

4. Save data to database:

```
conn.commit()
```

14.3.8 Query data

1. Select all records from an author:

```
def select_all_records_from_author(cursor, author):
    print(f"All books from {author}:")
    sql = "SELECT * FROM books WHERE author=?"
    cursor.execute(sql, [author])
    print(cursor.fetchall())  # or use fetchone()
```

For the `print` output, we use a formatted string literal or f-string by prefixing it with an `f`.

2. Select all records sorted by author:

```
def select_all_records_sorted_by_author(cursor):
    print("Listing of all books sorted by author:")
    for row in cursor.execute("SELECT rowid, * FROM books ORDER BY author"):
        print(row)
```

3. Select titles containing Python:

```
def select_using_like(cursor, text):
    print(f"All books with {text} in the title:")
    sql = f"""
    SELECT * FROM books
    WHERE title LIKE '{text}%'"""
    cursor.execute(sql)
    print(cursor.fetchall())
```

4. Finally, the data can be queried with:

```
select_all_records_from_author(cursor, author="Veit Schiele")
select_all_records_sorted_by_author(cursor)
select_using_like(cursor, text="Python")
```

```
All books from Veit Schiele:
[(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')]
Listing of all books sorted by author:
(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')
All books with Python in the title:
[(1, 'Python basics', 'en', 'Veit Schiele', 'BSD-3-Clause', '2021-10-28')]
```

14.3.9 Update data

1. Change a license

```
def update_license(old_name, new_name):
    conn = sqlite3.connect("library.db")
    cursor = conn.cursor()
    sql = f"""
    UPDATE books
    SET license = '{new_name}'
    WHERE license = '{old_name}'
    """
    cursor.execute(sql)
    conn.commit()
```

2. Calling the method:

```
update_license(old_name="BSD", new_name="BSD-3-Clause")
```

14.3.10 Delete data

1. Delete all books in a specific language:

```
def delete_by_language(language):
    conn = sqlite3.connect("library.db")
    cursor = conn.cursor()

    sql = f"""
    DELETE FROM books
    WHERE language = '{language}'
    """
    cursor.execute(sql)
    conn.commit()
```

2. Call the method with the parameter of the language to be deleted:

```
delete_by_language(language="de")
```

14.3.11 Normalising the data

Normalisation is the division of attributes or table columns into several relations or tables so that no redundancies are included.

Example

In the following example, we normalise the language in which the books were published.

1. To do this, we first create a new table `languages` with the columns `id` and `language_code`:

```
cursor.execute(
    """CREATE TABLE languages
                (id INTEGER PRIMARY KEY AUTOINCREMENT,
                language_code VARCHAR(2))"""
```

2. Then we create the values `de` and `en` in this table:

```
cursor.execute(
    """INSERT INTO languages (language_code)
                VALUES ('de')"""
)

cursor.execute(
    """INSERT INTO languages (language_code)
```

3. Since SQLite does not support `MODIFY COLUMN`, we now create a temporary table `temp` with all columns from `books` and a column `language_code` that uses the column `id` from the `languages` table as a foreign key:

```
cursor.execute(
    """CREATE TABLE "temp" (
                "id" INTEGER,
                "title" TEXT,
                "language_code" INTEGER REFERENCES languages(id),
```

(continues on next page)

(continued from previous page)

```
                    "language" TEXT,
                    "author" TEXT,
                    "license" TEXT,
                    "release_date" DATE,
                    PRIMARY KEY("id" AUTOINCREMENT)
                    )"""
```

4. Now we transfer the values from the `books` table to the `temp` table:

```
cursor.execute(
    """INSERT INTO temp (title,language,author,license,release_date)
                    SELECT title,language,author,license,release_date FROM books"""
```

5. Transfer the specification of the language in `books` as the `id` of the data records from the `languages` table to `temp`.

```
cursor.execute(
    """UPDATE temp
                    SET language_code = 1
                    WHERE language = 'de'"""
)
```

6. Now we can delete the `languages` column in the `temp` table:

```
cursor.execute("""ALTER TABLE temp DROP COLUMN language""")
```

Note: `DROP COLUMN` can only be used from Python versions from 3.8 that were released after 27 April 2021.

With older Python versions, another table would have to be created that no longer contains the languages column and then the data records from `temp` would have to be inserted into this table.

---

7. The `books` table can now also be deleted:

```
cursor.execute("""DROP TABLE books""")
```

8. And finally, the `temp` table can be renamed `books`:

```
cursor.execute("""ALTER TABLE temp RENAME TO books""")
```

14.3.12 Query normalised data

1. Query all books sorted by `language_id` and `title`:

```
def select_all_records_ordered_by_language_number(cursor):
    print("All books ordered by language id and title:")
    for row in cursor.execute(
        """SELECT language_code, author, title FROM books
                                ORDER BY language_code,title"""
    ):
        print(row)
```

```
All books ordered by language id and title:
(1, 'Veit Schiele', 'Jupyter Tutorial')
(2, 'Veit Schiele', 'Jupyter Tutorial')
(2, 'Veit Schiele', 'PyViz Tutorial')
(2, 'Veit Schiele', 'Python basics')
```

2. In order to receive not only the ID of the languages but also the corresponding language codes, a connection to the language codes stored there is established with `JOIN` via the `id` column in the `languages` table:

```
def select_all_records_ordered_by_language_code(cursor):
    print("All books ordered by language code and title:")
    for row in cursor.execute(
        """SELECT languages.language_code, books.author, books.title
                                FROM books
                                JOIN languages ON (books.language_code = languages.
↪id)
                                ORDER BY languages.language_code,title"""
    ):
        print(row)
```

```
All books ordered by language code and title:
('de', 'Veit Schiele', 'Jupyter Tutorial')
('en', 'Veit Schiele', 'Jupyter Tutorial')
('en', 'Veit Schiele', 'PyViz Tutorial')
('en', 'Veit Schiele', 'Python basics')
```

14.3.13 The psycopg module

1. Install the psycopg module

```
$ python3 -m pip install psycopg
Collecting psycopg
  Downloading psycopg-3.0.1-py3-none-any.whl (140 kB)
     || 140 kB 3.4 MB/s
Installing collected packages: psycopg
Successfully installed psycopg-3.0.1
```

2. Import the psycopg module

```
import psycopg2
```

3. Create a database

```
conn = psycopg2.connect(dbname="my_db", user="username")
cursor = conn.cursor()
```

4. Query the database

```
cursor.execute("SELECT * FROM my_table")
row = cursor.fetchone()
```

5. Close cursor and connection

```
11 cursor.close()
12 conn.close()
```

CHAPTER

FIFTEEN

DATACLASSES

dataclasses were introduced in Python 3.7 and are a special shortcut with which we can create classes that store data. Python offers a special *decorator* if we want to create such a class.

Note: For table data I generally use pandas Series or DataFrames and if I need to store matrices with numbers I use Numpy.

Let's say we want to store a class that represents an item with `summary`, `owner`, `state` and `id`. We can define such a class with:

```
>>> from dataclasses import dataclass
>>> @dataclass
... class Item:
...     summary: str = None
...     owner: str = None
...     state: str = "todo"
...     id: int = None
```

The `@dataclass` decorator creates the `__init__` and `__repr__` methods. If I display the instance of the class, I get the class name and the attributes:

```
>>> i1
Item(summary='My first item', owner='veit', state='todo', id=1)
```

In general, data classes are used as syntactic sugar for creating classes that store data. You can add extra functionality to your classes by defining methods. We will add a method to the class that creates an Item object from a *Dict*:

```
>>> @dataclass
... class Item:
...
...     @classmethod
...     def from_dict(cls, d):
...         return Item(**d)
...
>>> item_dict = {"summary": "My first item", "owner": "veit", "state": "todo", "id": 1}
>>> Item.from_dict(item_dict)
Item(summary='My first item', owner='veit', state='todo', id=1)
```

CHAPTER

SIXTEEN

TESTING

Basically, a distinction is made between static and dynamic test procedures.

Static test procedures
are used to check the source code, but it's not executed. They are divided into

- reviews and
- static program analysis

  There are several Python packages that can help you with static program analysis, including flake8, Pysa and Wily.

Dynamic testing
are used to find errors when executing the source code. A distinction is made between whitebox and backbox tests.

Whitebox tests
are developed with knowledge of the source code and the software structure. In Python, various modules are available:

Unittest
supports you in automating tests.

Mock
allows you to create and use mock objects.

Doctest
allows you to test tests written in Python docstrings.

tox
allows you to test in different environments.

Blackbox tests
are developed without knowledge of the source code. In addition to *Unittest*, *Hypothesis* can also be used in Python for such tests.

See also:

- Python Testing and Continuous Integration

16.1 Unittest

unittest supports you in test automation with shared setup and tear-down code as well as aggregation and independence of tests.

It provides the following test concepts:

Test Case
tests a single scenario.

Test Fixture
is a consistent test environment.

See also:

- pytest fixtures
- About fixtures
- Fixtures reference
- How to use fixtures

Test Suite
is a collection of several *test cases*.

Test Runner
runs through a *Test Suite* and displays the results.

16.1.1 Example

Suppose you have implemented the following add method in the `test_arithmetic.py` module:

```
def add(x, y):
    """
    >>> add(7,6)
    13
    """
    return x + y
```

... then you can test this method with a Unittest.

1. To do this, you must first import your module and the unittest module:

```
import unittest
class TestArithmetic(unittest.TestCase):
```

2. Then you can write a test method that illustrates your addition method:

```
class TestArithmetic(unittest.TestCase):
    def test_addition(self):
        self.assertEqual(arithmetic.add(7, 6), 13)

```

3. In order to import the unittests into other modules, you should add the following lines:

```
if __name__ == "__main__":
    unittest.main()
```

4. Finally, all tests in `test_arithmetic.py` can be executed:

```
$ bin/python test_arithmetic.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

```
C:> python test_arithmetic.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

   … or a little more detailed:

```
$ python test_arithmetic.py -v
test_addition (__main__.TestArithmetic) ... ok
test_division (__main__.TestArithmetic) ... ok
test_multiplication (__main__.TestArithmetic) ... ok
test_subtraction (__main__.TestArithmetic) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

```
C:> Scripts\python test_arithmetic.py -v
test_addition (__main__.TestArithmetic) ... ok
test_division (__main__.TestArithmetic) ... ok
test_multiplication (__main__.TestArithmetic) ... ok
test_subtraction (__main__.TestArithmetic) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

See also:

- unittest — Unit testing framework

16.2 Example: Testing an SQLite database

1. To test whether the database `library.db` was created with `create_db.py`, we import `../save-data/create_db.py` and os in addition to sqlite3 and unittest:

```
import os
import sqlite3
import unittest

import create_db
```

2. Then we first define a test class `TestCreateDB`:

```
class TestCreateDB(unittest.TestCase):
```

3. In it we then define the test method `test_db_exists`, in which we use `assert` to assume that the file exists in os.path:

```
    def test_db_exists(self):
        assert os.path.exists("library.db")
```

4. Now we also check whether the `books` table was created. For this we try to create the table again and expect with `assertRaises` that `sqlite` is terminated with an `OperationalError`:

```
    def test_table_exists(self):
        with self.assertRaises(sqlite3.OperationalError):
            create_db.cursor.execute("CREATE TABLE books(title text)")
```

5. We do not want to carry out further tests on a database in the file system but in an SQLite database in the working memory:

```
class TestCommands(unittest.TestCase):
    def setUp(self):
        self.conn = sqlite3.connect(":memory:")
        cursor = self.conn.cursor()
```

See also:

You can find more examples for testing your SQLite database functions in the SQLite test suite test_sqlite3.

16.3 Doctest

The Python module doctest checks whether the tests specified in a docstring are fulfilled.

1. In `arithmetic.py` you can add the following docstring:

```
def divide(x, y):
    """Divides the first parameter by the second
    >>> x, y, z = 7, -6.0, 0
    >>> divide(x, y)
    -1.1666666666666667
    >>> divide(x, z)
    Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
      File "<stdin>", line 1, in <module>
    ZeroDivisionError: division by zero
    """
```

2. Then you can test it with:

```
$ python -m doctest test/arithmetic.py -v
Trying:
    add(7,6)
Expecting:
    13
ok
Trying:
    x, y, z = 7, -6.0, 0
Expecting nothing
ok
Trying:
    divide(x, y)
Expecting:
    -1.1666666666666667
ok
Trying:
    divide(x, z)
Expecting:
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    ZeroDivisionError: division by zero
ok
Trying:
    multiply(7,6)
Expecting:
    42
ok
Trying:
    subtract(7,6)
Expecting:
    1
ok
1 items had no tests:
    arithmetic
4 items passed all tests:
   1 tests in arithmetic.add
   3 tests in arithmetic.divide
   1 tests in arithmetic.multiply
   1 tests in arithmetic.subtract
6 tests in 5 items.
6 passed and 0 failed.
Test passed.
```

```
C:> Scripts\python -m doctest arithmetic.py -v
Trying:
    add(7,6)
```

(continues on next page)

(continued from previous page)

```
Expecting:
    13
ok
Trying:
    x, y, z = 7, -6.0, 0
Expecting nothing
ok
Trying:
    divide(x, y)
Expecting:
    -1.1666666666666667
ok
Trying:
    divide(x, z)
Expecting:
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    ZeroDivisionError: division by zero
ok
Trying:
    multiply(7,6)
Expecting:
    42
ok
Trying:
    subtract(7,6)
Expecting:
    1
ok
1 items had no tests:
    arithmetic
4 items passed all tests:
   1 tests in arithmetic.add
   3 tests in arithmetic.divide
   1 tests in arithmetic.multiply
   1 tests in arithmetic.subtract
6 tests in 5 items.
6 passed and 0 failed.
Test passed.
```

3. So that the doctests can also be imported into other modules, you should add the following lines:

```
if __name__ == "__main__":
    import doctest

    doctest.testmod(verbose=True)
```

16.4 Hypothesis

Hypothesis is a library that allows you to write tests that are parameterised from a source of examples. It then generates simple and understandable examples that can be used to make your tests fail and find bugs with little effort.

1. Install Hypothesis:

```
$ bin/python -m pip install hypothesis
```

```
C:> Scripts\python -m pip install hypothesis
```

   Alternatively, Hypothesis can also be installed with extensions, for example:

```
$ bin/python -m pip install hypothesis[numpy,pandas]
```

```
C:> Scripts\python -m pip install hypothesis[numpy,pandas]
```

2. Write a test:

   1. Imports:

```
import pytest
from hypothesis import given
from hypothesis.strategies import floats, lists
```

   2. Test:

```
@given(lists(floats(allow_nan=False, allow_infinity=False), min_size=1))
def test_mean(ls):
    mean = sum(ls) / len(ls)
    assert min(ls) <= mean <= max(ls)
```

3. Perform test:

```
$ bin/python -m pytest test_hypothesis.py
============================= test session starts ==============================
platform darwin -- Python 3.9.7, pytest-6.2.5, py-1.10.0, pluggy-1.0.0
rootdir: /Users/veit/cusy/trn/python-basics/docs/test
plugins: hypothesis-6.23.2
collected 1 item

test_hypothesis.py F                                                     [100%]

=================================== FAILURES ===================================
_____________________________________ test_mean _____________________________________

    @given(lists(floats(allow_nan=False, allow_infinity=False), min_size=1))
>   def test_mean(ls):

test_hypothesis.py:6:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

ls = [9.9792015476736e+291, 1.7976931348623157e+308]
```

(continues on next page)

(continued from previous page)

```
    @given(lists(floats(allow_nan=False, allow_infinity=False), min_size=1))
    def test_mean(ls):
        mean = sum(ls) / len(ls)
>       assert min(ls) <= mean <= max(ls)
E       assert inf <= 1.7976931348623157e+308
E        +  where 1.7976931348623157e+308 = max([9.9792015476736e+291, 1.
↪7976931348623157e+308])

test_hypothesis.py:8: AssertionError
---------------------------------- Hypothesis ----------------------------------
Falsifying example: test_mean(
    ls=[9.9792015476736e+291, 1.7976931348623157e+308],
)
============================ short test summary info ============================
FAILED test_hypothesis.py::test_mean - assert inf <= 1.7976931348623157e+308
============================== 1 failed in 0.44s ===============================
```

```
C:> Scripts\python -m pytest test_hypothesis.py
============================= test session starts ==============================
platform win32 -- Python 3.9.7, pytest-6.2.5, py-1.10.0, pluggy-1.0.0
rootdir: C:\Users\veit\python-basics\docs\test
plugins: hypothesis-6.23.2
collected 1 item

test_hypothesis.py F                                                     [100%]

=================================== FAILURES ===================================
_____________________________________ test_mean _____________________________________

    @given(lists(floats(allow_nan=False, allow_infinity=False), min_size=1))
>   def test_mean(ls):

test_hypothesis.py:6:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

ls = [9.9792015476736e+291, 1.7976931348623157e+308]

    @given(lists(floats(allow_nan=False, allow_infinity=False), min_size=1))
    def test_mean(ls):
        mean = sum(ls) / len(ls)
>       assert min(ls) <= mean <= max(ls)
E       assert inf <= 1.7976931348623157e+308
E        +  where 1.7976931348623157e+308 = max([9.9792015476736e+291, 1.
↪7976931348623157e+308])

test_hypothesis.py:8: AssertionError
---------------------------------- Hypothesis ----------------------------------
Falsifying example: test_mean(
    ls=[9.9792015476736e+291, 1.7976931348623157e+308],
)
============================ short test summary info ============================
FAILED test_hypothesis.py::test_mean - assert inf <= 1.7976931348623157e+308
```

(continues on next page)

(continued from previous page)

```
============================== 1 failed in 0.44s ===============================
```

See also:

Hypothesis for the Scientific Stack

16.5 pytest

pytest is an alternative to Python's *Unittest* module that simplifies testing even further.

16.5.1 Features

- More detailed information about failed `assert` statements
- Automatic detection of test modules and functions
- Modular fixtures for the management of small or parameterised, long-lived test resources
- Can also execute unit tests without presets
- Extensive plug-in architecture, with over 800 external plug-ins

16.5.2 Installation

You can install pytest in *virtual environments* <*virtuelle-umgebungen*> with:

```
$ python -m pip install pytest
Collecting pytest
...
Successfully installed attrs-21.2.0 iniconfig-1.1.1 pluggy-1.0.0 py-1.10.0 pytest-6.2.5␣
↪toml-0.10.2
```

```
C:> python -m pip install pytest
Collecting pytest
...
Successfully installed attrs-21.2.0 iniconfig-1.1.1 pluggy-1.0.0 py-1.10.0 pytest-6.2.5␣
↪toml-0.10.2
```

Examples

You can simply create a file `test_one.py` with the following content:

```
1 def test_sorted():
2     assert sorted([4, 2, 1, 3]) == [1, 2, 3, 4]
```

The `test_sorted()` function is recognised by pytest as a test function because it starts with `test_` and is in a file that starts with `test_`. When the test is executed, the `assert` statement determines whether the test succeeded or failed. `assert` is a Python built-in keyword and raises an `assertionError` exception if the expression after `assert` is false. Any uncaught exception thrown within a test will cause the test to fail.

Execute pytest

```
$ cd docs/test/pytest
$ pytest test_one.py
============================= test session starts ==============================
...
collected 1 item

test_one.py .                                                            [100%]

============================== 1 passed in 0.00s ===============================
```

The dot after `test_one.py` means that a test has been performed and passed. `[100%]` is a percentage display that indicates how many tests of the test session have been performed so far. As there is only one test, one test corresponds to 100% of the tests. If you need more information, you can use `-v` or `--verbose`:

```
$ pytest -v test_one.py
============================= test session starts ==============================
...
collected 1 item

test_one.py::test_sorted PASSED                                          [100%]

============================== 1 passed in 0.00s ===============================
```

`test_two.py` on the other hand, fails:

```
$ pytest test_two.py
collected 1 item

test_two.py F                                                            [100%]

=================================== FAILURES ===================================
_________________________________ test_failing _________________________________

    def test_failing():
>       assert sorted([4, 2, 1, 3]) == [0, 1, 2, 3]
E       assert [1, 2, 3, 4] == [0, 1, 2, 3]
E         At index 0 diff: 1 != 0
E         Use -v to get more diff

test_two.py:2: AssertionError
=========================== short test summary info ============================
FAILED test_two.py::test_failing - assert [1, 2, 3, 4] == [0, 1, 2, 3]
============================== 1 failed in 0.03s ===============================
```

The failed test, `test_in`, gets its own section to show us why it failed. And `pytest` tells us exactly what the first error is. This additional section is called traceback. That's already a lot of information, but there's a line that says we get the full diff with `-v`. Let's do that:

```
$ pytest -v test_two.py
============================= test session starts ==============================
...
```

(continues on next page)

(continued from previous page)

```
collected 1 item

test_two.py::test_failing FAILED                                         [100%]

================================== FAILURES ===================================
_____________________________ test_failing ______________________________

    def test_failing():
>       assert sorted([4, 2, 1, 3]) == [0, 1, 2, 3]
E       assert [1, 2, 3, 4] == [0, 1, 2, 3]
E         At index 0 diff: 1 != 0
E         Full diff:
E         - [0, 1, 2, 3]
E         ?  ---
E         + [1, 2, 3, 4]
E         ?         +++

test_two.py:2: AssertionError
=========================== short test summary info ============================
FAILED test_two.py::test_failing - assert [1, 2, 3, 4] == [0, 1, 2, 3]
============================== 1 failed in 0.03s ===============================
```

`pytest` adds + and - signs to show us exactly the differences.

So far we have run `pytest` with the command `pytest` *`FILE`*`.py`. Now let's run `pytest` in a few more ways. If you don't specify any files or directories, pytest will look for tests in the current working directory and subdirectories; more specifically, it will look for `.py` files that start with `test_` or end with `_test`. If you start pytest in the directory `docs/test/pytest` without options, two files with tests will be run:

```
$ pytest --tb=no
============================= test session starts ==============================
...

test_one.py .                                                            [ 50%]
test_two.py F                                                            [100%]

=========================== short test summary info ============================
FAILED test_two.py::test_failing - assert [1, 2, 3, 4] == [0, 1, 2, 3]
========================== 1 failed, 1 passed in 0.00s =========================
```

I have also used the `--tb=no` option to disable traceback as we don't really need the full output at the moment.

We can also specify a test function within a test file to be executed by adding `::test_`*`name`* to the file name:

```
$ pytest -v test_one.py::test_sorted
============================= test session starts ==============================
...
collected 1 item

test_one.py::test_sorted PASSED                                          [100%]

============================== 1 passed in 0.00s ===============================
```

Test results

The possible results of a test function include

PASSED (.)
: The test was performed successfully.

FAILED (F)
: The test was not performed successfully.

SKIPPED (s)
: The test was skipped.

XFAIL (x)
: The test should not pass, but was performed and failed.

XPASS (X)
: The test was marked `xfail`, but it ran and passed.

ERROR (E)
: An exception occurred during the execution of a *Test fixtures*, but not during the execution of a test function.

Writing test functions

`assert` statements

When writing test functions, the normal pytest `assert` statement is your most important tool. The simplicity of this statement leads many developers to favour pytest over other frameworks. Below is a list of some of *Unittest*'s `assert` forms and `assert` helper functions:

pytest	unittest
`assert something`	`assertTrue(something)`
`assert not something`	`assertFalse(something)`
`assert x == y`	`assertEqual(x, y)`
`assert x != y`	`assertNotEqual(x, y)`
`assert x <= y`	`assertLessEqual(x, y)`
`assert x is None`	`assertIsNone(x)`
`assert x is not None`	`assertIsNotNone(x)`

With pytest you can use `assert EXPRESSION` with any expression. If the expression would evaluate to `False` when converted to a boolean value, the test would fail.

pytest includes a function called `assert rewriting` that intercepts `assert` calls and replaces them with something that can tell you more about why your assumptions failed. Let's see how helpful this rewriting is by looking at a failed `assert` test:

```
def test_equality_fails():
    i1 = Item("do something", "veit")
    i2 = Item("do something else", "veit")
    assert i1 == i2
```

This test fails, but the traceback information is interesting:

```
$ pytest tests/test_item_fails.py
============================= test session starts ==============================
...
collected 1 item

tests/test_item_fails.py F                                               [100%]

=================================== FAILURES ===================================
_____________________________ test_equality_fails ______________________________

    def test_equality_fails():
        i1 = Item("do something", "veit")
        i2 = Item("do something else", "veit.schiele")
>       assert i1 == i2
E       AssertionError: assert Item(summary=...odo', id=None) == Item(summary=...odo',␣
↪id=None)
E
E         Omitting 1 identical items, use -vv to show
E         Differing attributes:
E         ['summary', 'owner']
E
E         Drill down into differing attribute summary:
E           summary: 'do something' != 'do something else'...
E
E         ...Full output truncated (8 lines hidden), use '-vv' to show

tests/test_item_fails.py:7: AssertionError
=========================== short test summary info ============================
FAILED tests/test_item_fails.py::test_equality_fails - AssertionError: assert␣
↪Item(summary=...odo', id=None) == Item(summary=...od...
============================== 1 failed in 0.03s ===============================
```

That's a lot of information:

For each failed test, the exact line of the error is displayed with a > pointing to the error.

The `E` lines show you additional information about the `assert` error so you can figure out what went wrong. I intentionally entered two mismatches in `test_equality_fails()`, but only the first one was displayed. Let's try again with the `-vv` option as suggested in the error message:

```
$ pytest -vv tests/test_item_fails.py
============================= test session starts ==============================
...
collected 1 item

tests/test_item_fails.py::test_equality_fails FAILED                     [100%]

=================================== FAILURES ===================================
_____________________________ test_equality_fails ______________________________

    def test_equality_fails():
        i1 = Item("do something", "veit")
        i2 = Item("do something else", "veit.schiele")
>       assert i1 == i2
```

(continues on next page)

(continued from previous page)

```
E       AssertionError: assert Item(summary='do something', owner='veit', state='todo',␣
↪id=None) == Item(summary='do something else', owner='veit.schiele', state='todo',␣
↪id=None)
E
E         Matching attributes:
E         ['state']
E         Differing attributes:
E         ['summary', 'owner']
E
E         Drill down into differing attribute summary:
E           summary: 'do something' != 'do something else'
E           - do something else
E           ?             -----
E           + do something
E
E         Drill down into differing attribute owner:
E           owner: 'veit' != 'veit.schiele'
E           - veit.schiele
E           + veit

tests/test_item_fails.py:7: AssertionError
=========================== short test summary info ============================
FAILED tests/test_item_fails.py::test_equality_fails - AssertionError: assert␣
↪Item(summary='do something', owner='veit', state='to...
============================== 1 failed in 0.03s ===============================
```

pytest has listed exactly which attributes match and which do not. The exact deviations were also highlighted.

For comparison, we can see what Python displays for `assert` errors. To be able to call the test directly from Python, we need to add a block at the end of `tests/test_item_fails.py`:

```
if __name__ == "__main__":
    test_equality_fails()
```

If we now run the test with Python, we get the following result:

```
python tests/test_item_fails.py
Traceback (most recent call last):
  File "tests/test_item_fails.py", line 11, in <module>
    test_equality_fails()
  File "tests/test_item_fails.py", line 7, in test_equality_fails
    assert i1 == i2
           ^^^^^^^^
AssertionError
```

That doesn't tell us much. The pytest output gives us much more information about why our assumptions failed.

Failing with `pytest.fail()` and exceptions

Failing assertions is the main way that tests fail. But this is not the only way. A test also fails if there is an uncaught *Exceptions*. This can happen when

- an `assert` statement fails, resulting in an `AssertionError` exception,
- the test code calls `pytest.fail()`, which leads to an exception, or
- another exception is thrown.

Although any exception can cause a test to fail, I prefer to use `assert`. In rare cases where `assert` is not appropriate, I usually use `pytest.fail()`.

Here is an example of using pytest's `fail()` function to explicitly fail a test:

```
def test_with_fail():
    i1 = Item("do something", "veit")
    i2 = Item("do something else", "veit.schiele")
    if i1 != i2:
        pytest.fail("The items are not identical!")
```

The output is as follows:

```
pytest tests/test_item_fails.py
============================= test session starts ==============================
...
collected 1 item

tests/test_item_fails.py F                                               [100%]

=================================== FAILURES ===================================
_________________________________ test_with_fail _________________________________

    def test_with_fail():
        i1 = Item("do something", "veit")
        i2 = Item("do something else", "veit.schiele")
        if i1 != i2:
>           pytest.fail("The items are not identical!")
E           Failed: The items are not identical!

tests/test_item_fails.py:10: Failed
=========================== short test summary info ============================
FAILED tests/test_item_fails.py::test_with_fail - Failed: The items are not identical!
============================== 1 failed in 0.03s ===============================
```

When calling `pytest.fail()` or throwing an exception, we do not get the `assert` rewriting provided by pytest. However, there are useful occasions to use `pytest.fail()`, such as in an `assertion` utility.

Writing `assertion` helper functions

An `assertion` helper function is used to package a complicated `assertion` check. For example, the `Item` data class is set up so that two items with different IDs still report equality. If we want a stricter check, we could write a helper function called `assert_ident` as follows:

```
import pytest

from items import Item


def assert_ident(i1: Item, i2: Item):
    __tracebackhide__ = True
    assert i1 == i2
    if i1.id != i2.id:
        pytest.fail(f"The IDs do not match: {i1.id} != {i2.id}")


def test_ident():
    i1 = Item("something to do", id=42)
    i2 = Item("something to do", id=42)
    assert_ident(i1, i2)


def test_ident_fail():
    i1 = Item("something to do", id=42)
    i2 = Item("something to do", id=43)
    assert_ident(i1, i2)
```

The `assert_ident` function sets `__tracebackhide__ = True`. The result is that failed tests are not included in the traceback. The normal `assert i1 == i2` is then used to check everything except id for equality.

Finally, the IDs checked `pytest.fail()` are used to fail the test with a helpful message. Let's take a look at what this looks like after execution:

```
$ pytest tests/test_helper.py
============================= test session starts ==============================
...
collected 2 items

tests/test_helper.py .F                                                  [100%]

=================================== FAILURES ===================================
_____________________________ test_ident_fail ______________________________

    def test_ident_fail():
        i1 = Item("something to do", id=42)
        i2 = Item("something to do", id=43)
>       assert_ident(i1, i2)
E       Failed: The IDs do not match: 42 != 43

tests/test_helper.py:22: Failed
========================== short test summary info ===========================
FAILED tests/test_helper.py::test_ident_fail - Failed: The IDs do not match: 42 != 43
```

(continues on next page)

(continued from previous page)

```
========================= 1 failed, 1 passed in 0.03s ==========================
```

Testing for expected exceptions

We have looked at how any exception can cause a test to fail. But what if part of the code we are testing should raise an exception? For this we use `pytest.raises()` to test for expected exceptions. An example of this would be the Items API, which has an `ItemsDB` class that requires a path argument.

```
from items.api import ItemsDB


def test_db_exists():
    ItemsDB()
```

```
$ pytest --tb=short tests/test_db.py
============================= test session starts ==============================
...
collected 1 item

tests/test_db.py F                                                       [100%]

=================================== FAILURES ===================================
_____________________________ test_db_exists ______________________________
tests/test_db.py:5: in test_db_exists
    ItemsDB()
E   TypeError: ItemsDB.__init__() missing 1 required positional argument: 'db_path'
============================= short test summary info ============================
FAILED tests/test_db.py::test_db_exists - TypeError: ItemsDB.__init__() missing 1␣
↪required positional argument: 'db_p...
============================== 1 failed in 0.03s ===============================
```

Here I have used the shorter traceback format `--tb=short` because we don't need to see the full traceback to find out which exception was thrown.

The exception `TypeError` seems to make sense because the error occurs when trying to initialise the custom `ItemsDB` type. We can write a test to ensure that this exception is thrown, something like this:

```
import pytest

from items.api import ItemsDB


def test_db_exists():
    with pytest.raises(TypeError):
        ItemsDB()
```

The instruction with `pytest.raises(TypeError):` states that the next code block should throw a `TypeError` exception. If no exception or another exception is raised, the test fails.

We have just checked the type of the exception in `test_db_exists()`. We can also check if the message is correct, or any other aspect of the exception, such as additional parameters:

```
def test_db_exists():
    match_regex = "missing 1 .* positional argument"
    with pytest.raises(TypeError, match=match_regex):
        ItemsDB()
```

or

```
def test_db_exists():
    with pytest.raises(TypeError) as exc_info:
        ItemsDB()
    expected = "missing 1 required positional argument"
    assert expected in str(exc_info.value)
```

Structure test suite

You should ensure that assertions are kept at the end of test functions. This recommendation is so common that it has at least two names:

Arrange-Act-Assert (AAA)
: became popular as part of *test-driven development (TDD)*.

Given-When-Then (GWT)
: is used in the context of behaviour-driven development (BDD).

The division into these free phases has many advantages. This separates the parts

Given/Arrange
: The initial state. This is where you set up data or the environment to prepare the action.

When/Act
: An action is executed. This is the focus of the test – the behaviour that we want to ensure works correctly.

Then/Assert
: An expected result or end state should occur. At the end of the test, we make sure that the action has led to the expected behaviour.

A common counter-pattern is the *Arrange–Assert–Act–Assert–Act–Assert...* pattern, where a variety of actions followed by state or behavioural checks validate a workflow. This seems reasonable until the test fails. Any of the actions could have caused the failure, so the test doesn't focus on testing a particular behaviour. Or it could have been the setup in *Arrange* that caused the error. This nested `assert` pattern leads to tests that are difficult to debug and maintain. Sticking to * Given-When-Then* or *Arrange-Act-Assert* keeps the test focused and makes it more maintainable.

Let's apply this structure to one of our first tests as an example:

```
def test_equality_fail():
    # Given two item objects with known contents
    i1 = Item("do something", "veit")
    i2 = Item("do something else", "veit.schiele")
    # WHEN the two item objects are not identical
    if i1 != i2:
        # THEN the result will be a string
        pytest.fail("The items are not identical!")
```

The structure helps you to organise the test functions and focus on testing **one** behaviour. The structure also helps you to think of other test cases. Focusing on an initial state helps you to think of other states that might be relevant for testing the same action. Similarly, focusing on an ideal outcome helps you think of other possible outcomes, such as failure states or error states, that should also be tested with other test cases.

Grouping tests with classes

Up to now, we have written test functions within test modules in a file system directory. This structuring of the test code actually works quite well and is sufficient for many projects. However, pytest also allows us to group tests with classes. Let's take some of the test functions that relate to the equality of items and group them into a class:

```
class TestEquality:
    def test_equality(self):
        i1 = Item("do something", "veit", "todo", 42)
        i2 = Item("do something", "veit", "todo", 42)
        assert i1 == i2

    def test_equality_with_diff_ids(self):
        i1 = Item("do something", "veit", "todo", 42)
        i2 = Item("do something", "veit", "todo", 43)
        assert i1 == i2

    def test_inequality(self):
        i1 = Item("do something", "veit", "todo", 42)
        i2 = Item("do something else", "veit", "done", 42)
        assert i1 != i2
```

The code looks pretty much the same as before, with the exception that each method must have an initial `self` argument. We can now execute all these methods together by specifying the class:

```
$ pytest -v tests/test_classes.py::TestEquality
============================== test session starts ==============================
...
collected 3 items

tests/test_classes.py::TestEquality::test_equality PASSED                [ 33%]
tests/test_classes.py::TestEquality::test_equality_with_diff_ids PASSED  [ 66%]
tests/test_classes.py::TestEquality::test_inequality PASSED               [100%]

=============================== 3 passed in 0.00s ===============================
```

However, we can still call a single method:

```
$ pytest -v tests/test_classes.py::TestEquality::test_equality
============================== test session starts ==============================
...
collected 1 item

tests/test_classes.py::TestEquality::test_equality PASSED                [100%]

=============================== 1 passed in 0.00s ===============================
```

If you are familiar with *Object Orientation* and *class inheritance*, you can use hierarchies of test classes for inherited helper methods. I recommend that you use test classes sparingly and mainly for grouping, even in productive test code. If you go to too much trouble with test class inheritance, it will get confusing in the future.

Executing a subset of tests

In the previous section, we used test classes to execute a subset of tests. Executing a small group of tests is very handy when debugging, or if you want to limit the tests to a specific section of the codebase you are working on. pytest allows you to execute a subset of tests in different ways:

Subset	Syntax
All tests in one directory	`pytest path`
All tests in a module	`pytest path/test_module.py`
All tests in a class	`pytest path/test_module.py::TestClass`
Single test function	`pytest path/test_module.py::test_function`
Single test method	`pytest path/test_module.py::TestClass::test_method`
Tests that correspond to a name pattern	`pytest -k pattern`
Tests by marker	siehe *Markers*

Whether `pytest` finds your test code depends on the naming:

- Test files should be named `test_something.py` or `something_test.py`.
- Test methods and functions should be named `test_something`.
- Test classes should be named `TestSomething`.

Tip: Use a directory structure that corresponds to the way you want to run your code, because it is easy to run a complete subdirectory. This way you can divide features and functions or use subsystems as a basis or orientate yourself on the code structure.

You can also use `-k pattern` to filter directories, classes or test prefixes, for example all tests of class `TestEquality`.

```
$ pytest -v -k TestEquality
============================== test session starts ==============================
...
collected 7 items / 4 deselected / 3 selected

test_classes.py::TestEquality::test_equality PASSED                      [ 33%]
test_classes.py::TestEquality::test_equality_with_diff_ids PASSED        [ 66%]
test_classes.py::TestEquality::test_inequality PASSED                    [100%]

======================= 3 passed, 4 deselected in 0.00s =======================
```

or all tests with `equality` in the name:

```
pytest -v --tb=no -k equality
============================== test session starts ==============================
...
collected 7 items / 3 deselected / 4 selected

test_classes.py::TestEquality::test_equality PASSED                      [ 25%]
test_classes.py::TestEquality::test_equality_with_diff_ids PASSED        [ 50%]
test_classes.py::TestEquality::test_inequality PASSED                    [ 75%]
test_item_fail.py::test_equality_fail FAILED                             [100%]
```

(continues on next page)

(continued from previous page)

```
========================== short test summary info ============================
FAILED test_item_fail.py::test_equality_fail - Failed: The items are not identical!
================== 1 failed, 3 passed, 3 deselected in 0.01s ===================
```

Unfortunately, one of these is our error example. We can remove it by expanding the expression:

```
$ pytest -v --tb=no -k "equality and not equality_fail"
============================== test session starts ==============================
...
collected 7 items / 4 deselected / 3 selected

test_classes.py::TestEquality::test_equality PASSED                       [ 33%]
test_classes.py::TestEquality::test_equality_with_diff_ids PASSED         [ 66%]
test_classes.py::TestEquality::test_inequality PASSED                     [100%]

======================= 3 passed, 4 deselected in 0.00s ========================
```

The keywords `and`, `not`, `or` and `()` are allowed to create complex expressions. Here is a test run of all tests with or "ids" in the name, but not in the "TestEquality" class:

```
$ pytest -v --tb=no -k "(inequality or id) and not _fail"
============================== test session starts ==============================
...
collected 7 items / 4 deselected / 3 selected

test_classes.py::TestEquality::test_equality_with_diff_ids PASSED         [ 33%]
test_classes.py::TestEquality::test_inequality PASSED                     [ 66%]
test_helper.py::test_ident PASSED                                         [100%]

======================= 3 passed, 4 deselected in 0.00s ========================
```

The `-k` keyword option, together with `and`, `not` and `or`, offers great flexibility when selecting the tests you want to run. This proves to be very helpful when troubleshooting or developing new tests.

Tip: It is a good idea to use quotation marks when selecting a test to run as the hyphens, brackets and spaces can confuse the shells.

Test fixtures

Now that you have used pytest to write and execute test functions, let's move on to *fixtures*, which are essential for structuring test code for almost any non-trivial software system. Fixtures are functions that are executed by pytest before (and sometimes after) the actual test functions. The code in the fixture can do whatever you want. You can use fixtures to get a data set for the tests to work with. You can use fixtures to put a system into a known state before a test is executed. Fixtures are also used to provide data for multiple tests.

In this chapter, you will learn how to create and work with fixtures. You will learn how to structure fixtures to store both setup and teardown code. You will use `scope` to run fixtures once across many tests and learn how tests can use multiple fixtures. You will also learn how to track code execution through fixtures and test code.

But before you familiarise yourself with fixtures and use them to test Items, let's take a look at a small example fixture and learn how fixtures and test functions are connected.

First steps with fixtures

Here is a simple fixture that returns a number:

```
import pytest


@pytest.fixture()
def some_data():
    """The answer to the ultimate question"""
    return 42


def test_some_data(some_data):
    """Use fixture return value in a test."""
    assert some_data == 42
```

The `@pytest.fixture()` *decorator* is used to tell pytest that a function is a fixture. If you include the fixture name in the parameter list of a test function, pytest knows that the function should be executed before the test is run. Fixtures can perform work and also return data to the test function. In this case, `@pytest.fixture()` decorates the function `some_data()`. The test `test_some_data()` has the name of the fixture, `some_data()` as a parameter. pytest recognises this and searches for a fixture with this name.

Test fixtures in pytest refer to the mechanism that allows the separation of preparation for and cleanup after code from your test functions. pytest handles exceptions during fixtures differently than during a test function. An `Exception` or an `assert` error or a `pytest.fail()` call that occurs during the actual test code leads to a `Fail` result. During a fixture, however, the test function is reported as an error. This distinction is helpful when troubleshooting if a test has failed. If a test ends with a fail, the error is somewhere in the test function; if a test ends with an error, the error is somewhere in a fixture.

Using fixtures for setup and teardown

Fixtures will be a great help when testing the Items application. The Items application consists of an API that does most of the work and logic, a lean CLI and a database. Handling the database is an area where fixtures will be of great help:

```
from pathlib import Path
from tempfile import TemporaryDirectory

import items


def test_empty():
    with TemporaryDirectory() as db_dir:
        db_path = Path(db_dir)
        db = items.ItemsDB(db_path)
        count = db.count()
        db.close()
        assert count == 0
```

To be able to call `count()`, we need a database object, which we obtain by calling `items.ItemsDB(db_path)()`. The `items.ItemsDB()` function returns an `ItemsDB` object. The parameter `db_path` must be a `pathlib.Path` object that points to the database directory. For testing, a temporary directory that we obtain with `tempfile.TemporaryDirectory()` works.

However, this test function contains some problems: The code to set up the database before we call `count()` is not really what we want to test. Also, the `assert` statement cannot be done before calling `db.close()`, because if the `assert` statement fails, the database connection will no longer be closed. These problems can be solved with pytest fixture:

```
import pytest


@pytest.fixture()
def items_db():
    with TemporaryDirectory() as db_dir:
        db_path = Path(db_dir)
        db = items.ItemsDB(db_path)
        yield db
        db.close()


def test_empty(items_db):
    assert items_db.count() == 0
```

The test function itself is now much easier to read, as we have outsourced the entire database initialisation to a fixture called `items_db`. The `items_db` fixture prepares the test by providing the database and then outputting the database object. Only then is the test executed. And only after the test has run is the database closed again.

Fixture functions are executed before the tests that use them. If there is a `yield` in the function, it stops there, passes control to the tests and continues in the next line after the tests have been completed. The code above the `yield` is setup and the code after the `yield` is teardown. The teardown is guaranteed to be executed regardless of what happens during the tests.

In our example, `yield` takes place within a context manager with a temporary directory. This directory remains in place while the fixture is in use and the tests are running. At the end of the test, control is passed back to the fixture, `db.close()` can be executed and the `with` block can close access to the directory.

We can also use fixtures in several tests, for example in

```
def test_count(items_db):
    items_db.add_item(items.Item("something"))
    items_db.add_item(items.Item("something else"))
    assert items_db.count() == 2
```

`test_count()` uses the same `items_db` fixture. This time we take the empty database and add two items before checking the count. We can now use `items_db` for any test that requires a configured database. The individual tests, such as `test_empty()` and `test_count()`, can be kept smaller and focus on what we really want to test, rather than setup and teardown.

Show fixture execution with `--setup-show`

Now that we have two tests using the same fixture, it would be interesting to know in which order they are called. pytest offers the command line option `--setup-show`, which shows us the order of operations of tests and fixtures, including the setup and teardown phases of the fixtures:

```
$ pytest --setup-show tests/test_count.py
============================= test session starts ==============================
...
```

(continues on next page)

(continued from previous page)

```
collected 2 items

tests/test_count.py
        SETUP    F items_db
        tests/test_count.py::test_empty (fixtures used: items_db).
        TEARDOWN F items_db
        SETUP    F items_db
        tests/test_count.py::test_count (fixtures used: items_db).
        TEARDOWN F items_db

============================== 2 passed in 0.01s ===============================
```

We can see that our test is running, surrounded by the `SETUP` and `TEARDOWN` parts of the `items_db` fixture. The `F` in front of the fixture name indicates that the fixture is using the function scope, meaning that the fixture is called before each test function it uses, and then dismantled afterwards. Next, let's take a look at the functional scope.

Defining the scope of a fixture

Each fixture has a specific scope, which determines the order of execution of setup and teardown in relation to the execution of all test functions that use the fixture. The scope determines how often setup and teardown are executed when they are used by multiple test functions.

However, if setting up and connecting to the database or creating large data sets is time-consuming, you may not want to do this for every single test. We can change a range so that the slow part only happens once for multiple tests. Let's change the scope of our fixture so that the database is only opened once by adding `scope="module"` to the fixture decorator:

```
@pytest.fixture(scope="module")
def items_db():
    with TemporaryDirectory() as db_dir:
        db_path = Path(db_dir)
        db = items.ItemsDB(db_path)
        yield db
        db.close()
```

```
$ pytest --setup-show tests/test_count.py
============================== test session starts ==============================
...
collected 2 items

tests/test_count.py
    SETUP    M items_db
        tests/test_count.py::test_empty (fixtures used: items_db).
        tests/test_count.py::test_count (fixtures used: items_db).
    TEARDOWN M items_db

============================== 2 passed in 0.01s ===============================
```

We have saved this setup time for the second test function. By changing the module scope, any test in this module that uses the `items_db` fixture can use the same instance of it without incurring additional setup and teardown time.

However, the fixture parameter `scope` allows for more than just `module`:

scope values	Description
scope='functi	Default value. Is executed once per test function.
scope='class'	Executed once per test class, regardless of how many test methods the class contains.
scope='module	Executed once per module, regardless of how ny test functions or methods or other fixtures in the module use it.
scope='packag	Executed once per package or test directory, regardless of how many test functions or methods or other fixtures are used in the package.
scope='sessic	Executed once per session. All test methods and functions that use a fixture with session scope share a call for setup and teardown.

The scope is therefore determined when a fixture is defined and not at the point at which it is called. The test functions that use a fixture do not control how often a fixture is set up and dismantled.

For a fixture defined within a test module, the session and package scopes behave exactly like the module scopes. To be able to use these other scopes, we need to use a `conftest.py` file.

Sharing fixtures with `conftest.py`

You can insert fixtures into individual test files, but to share fixtures across multiple test files, you must use a `conftest.py` file either in the same directory as the test file that uses it or in a parent directory. The `conftest.py` file is optional. It is considered a local plugin by pytest and can contain hook functions and fixtures. Let's start by moving the `items_db` fixture from `test_count.py` to a `conftest.py` file in the same directory:

```
from pathlib import Path
from tempfile import TemporaryDirectory

import pytest

import items


@pytest.fixture(scope="session")
def items_db():
    """ItemsDB object connected to a temporary database"""
    with TemporaryDirectory() as db_dir:
        db_path = Path(db_dir)
        db = items.ItemsDB(db_path)
        yield db
        db.close()
```

Note: Fixtures can only depend on other fixtures in the same or a larger area. A fixture with a function scope can therefore depend on other fixtures with a function scope. A function scope fixture can also depend on `class`, `module` and `session` scope fixtures, but not vice versa.

Warning: Although `conftest.py` is a Python module, it should not be imported from test files. The `conftest.py` file is automatically read by pytest, so you do not need to import `conftest` anywhere.

Find where fixtures are defined

We have moved a fixture from the test module to a `conftest.py` file. We can have `conftest.py` files at really any level of our test directory. The tests can use any fixture that is in the same test module as a test function, or in a `conftest.py` file in the same directory, or at any level of the parent directory up to the root of the tests.

This creates a problem if you can't remember where a particular fixture is located and you want to see the source code. With `pytest --fixtures` we can display where the fixtures are defined:

```
pytest --fixtures
============================= test session starts ==============================
...
collected 10 items
cache -- .../_pytest/cacheprovider.py:532
    Return a cache object that can persist state between testing sessions.
...
tmp_path_factory [session scope] -- .../_pytest/tmpdir.py:245
    Return a :class:`pytest.TempPathFactory` instance for the test session.

tmp_path -- .../_pytest/tmpdir.py:260
    Return a temporary directory path object which is unique to each test
    function invocation, created as a sub directory of the base temporary
    directory.


---------------------- fixtures defined from tests.conftest ----------------------
items_db [session scope] -- conftest.py:10
    ItemsDB object connected to a temporary database


------------------ fixtures defined from tests.test_fixtures -------------------
some_data -- test_fixtures.py:5
    The answer to the ultimate question


============================== no tests ran in 0.00s ===============================
```

pytest shows us a list of all available fixtures that our test can use. This list contains a number of built-in fixtures, which we will look at in *Built-in fixtures*, as well as fixtures provided by *Plugins*. The fixtures found in `conftest.py` files are at the end of the list. If you specify a directory, pytest will list the fixtures that are available for tests in that directory. If you specify the name of a test file, pytest also includes the fixtures defined in the test modules.

The output of pytest contains

- the first line of the docstring of the fixture; by adding `-v`, the entire docstring is included
- the file and line number in which the fixture is defined
- the path if the fixture is not in the current directory

Note: We have to use `-v` for pytest 6.x to get the path and the line numbers. Only from pytest 7 onwards will these be added without any further option.

You can also use `--fixtures-per-test` to see which fixtures are used by each test and where the fixtures are defined:

```
pytest --fixtures-per-test test_count.py::test_empty
============================= test session starts ==============================
...
collected 1 item

------------------------ fixtures used by test_empty -------------------------
----------------------------- (test_count.py:5) ------------------------------
items_db -- conftest.py:10
    ItemsDB object connected to a temporary database

============================== no tests ran in 0.00s ===============================
```

In this example, we have specified a single test: `test_count.py::test_empty`. However, files or directories can also be specified.

Using multiple fixture levels

Our test code is still problematic at the moment, as both tests depend on the database being empty at the beginning. This problem becomes very clear when we add a third test:

```
$ pytest test_count.py::test_count2
============================= test session starts ==============================
...
collected 1 item

test_count.py .                                                          [100%]

============================== 1 passed in 0.00s ===============================
```

It works when executed individually, but not when executed after `test_count.py::test_count`:

```
$ pytest test_count.py
============================= test session starts ==============================
...
collected 3 items

test_count.py ..F                                                        [100%]

=================================== FAILURES ===================================
_____________________________________ test_count2 ______________________________________

items_db = <items.api.ItemsDB object at 0x103d3a390>

    def test_count2(items_db):
        items_db.add_item(items.Item("something different"))
>       assert items_db.count() == 1
E       assert 3 == 1
E        +  where 3 = <bound method ItemsDB.count of <items.api.ItemsDB object at␣
↪0x103d3a390>>()
E        +    where <bound method ItemsDB.count of <items.api.ItemsDB object at␣
↪0x103d3a390>> = <items.api.ItemsDB object at 0x103d3a390>.count
```

(continues on next page)

(continued from previous page)

```
test_count.py:15: AssertionError
============================ short test summary info ============================
FAILED test_count.py::test_count2 - assert 3 == 1
========================== 1 failed, 2 passed in 0.03s ==========================
```

There are three items in the database because the previous test already added two items before `test_count2` was executed. However, tests should not rely on the order of execution. `test_count2` only succeeds if it is executed alone, but fails if it is executed after `test_count`.

If we still want to try to work with an open database but start all tests with zero items in the database, we can do this by adding another fixture in `conftest.py`:

```
@pytest.fixture(scope="session")
def db():
    """ItemsDB object connected to a temporary database"""
    with TemporaryDirectory() as db_dir:
        db_path = Path(db_dir)
        db_ = items.ItemsDB(db_path)
        yield db_
        db_.close()


@pytest.fixture(scope="function")
def items_db(db):
    """ItemsDB object that's empty"""
    db.delete_all()
    return db
```

I have renamed the old `items_db` to `db` and moved it to the session area.

The `items_db` fixture has `db` in its parameter list, which means that it depends on the `db` fixture. In addition, `items_db` is `function`-orientated, which is a narrower scope than `db`. If fixtures depend on other fixtures, they can only use fixtures that have the same or a larger scope.

Let's see if it works:

```
$ pytest --setup-show test_count.py
============================= test session starts ==============================
...
collected 3 items

test_count.py
SETUP    S db
        SETUP    F items_db (fixtures used: db)
        test_count.py::test_empty (fixtures used: db, items_db).
        TEARDOWN F items_db
        SETUP    F items_db (fixtures used: db)
        test_count.py::test_count (fixtures used: db, items_db).
        TEARDOWN F items_db
        SETUP    F items_db (fixtures used: db)
        test_count.py::test_count2 (fixtures used: db, items_db).
        TEARDOWN F items_db
TEARDOWN S db
```

(continues on next page)

(continued from previous page)

```
============================== 3 passed in 0.00s ===============================
```

We see that the setup for db is done first and has the scope of the session (from the S). The setup for `items_db` happens next and before each test function call and has the scope of the function (from the F). In addition, all three tests are passed.

Using fixtures for multiple stages can provide incredible speed advantages and maintain test order independence.

Using multiple fixtures per test or fixture

Another way to use multiple fixtures is to use more than one in a function or fixture. For example, we can put some pre-planned items together to test them in one fixture:

```
@pytest.fixture(scope="session")
def items_list():
    """List of different Item objects"""
    return [
        items.Item("Add Python 3.12 static type improvements", "veit", "todo"),
        items.Item("Add tips for efficient testing", "veit", "wip"),
        items.Item("Update cibuildwheel section", "veit", "done"),
        items.Item("Add backend examples", "veit", "done"),
    ]
```

Dann können wir sowohl `empty_db` als auch `items_list` in `test_add.py` verwenden:

```
def test_add_list(items_db, items_list):
    expected_count = len(items_list)
    for i in items_list:
        items_db.add_item(i)
    assert items_db.count() == expected_count
```

And fixtures can also use several other fixtures:

```
@pytest.fixture(scope="function")
def populated_db(items_db, items_list):
    """ItemsDB object populated with 'items_list'"""
    for i in some_items:
        items_db.add_item(i)
    return items_db
```

The fixture `populated_db` must be in the function area, as it uses `items_db`, which is already in the `function` area. If you try to place `populated_db` in the `module` area or a larger area, pytest will issue an error. Don't forget that if you don't specify a range, you will get fixtures in the `function` area. Tests that require a populated database can now simply do this with

```
def populated(populated_db):
    assert populated_db.count() > 0
```

We have seen how different fixture scopes work and how different scopes can be used in different fixtures. However, you may need to define a scope at runtime. This is possible with dynamic scoping.

Set fixture scope dynamically

Let's assume we have set up the fixtures as they are now, with db in the `session` scope and `items_db` in the `function` scope. However, there is now a risk that the `items_db` fixture is empty because it calls `delete_all()`. We therefore want to create a way of setting up the database completely for each test function by dynamically defining the scope of the db fixture at runtime. To do this, we first change the scope of db in the `conftest.py` file:

```
@pytest.fixture(scope=db_scope)
def db():
    """ItemsDB object connected to a temporary database"""
    with TemporaryDirectory() as db_dir:
        db_path = Path(db_dir)
        db_ = items.ItemsDB(db_path)
        yield db_
        db_.close()
```

Instead of a specific scope, we have entered a function name: `db_scope`. Now we have to write this function:

```
def db_scope(fixture_name, config):
    if config.getoption("--fdb", None):
        return "function"
    return "session"
```

There are many ways in which we can find out which area we should use. In this case, I decided to use a new command line option `--fdb`. In order to use this new option with pytest, we need to write a hook function in the `conftest.py` file, which I will explain in more detail in *Plugins*:

```
def pytest_addoption(parser):
    parser.addoption(
        "--fdb",
        action="store_true",
        default=False,
        help="Create new db for each test",
    )
```

After all this, the default behaviour is the same as before, with db in the `session` scope:

```
$ pytest --setup-show test_count.py
============================= test session starts ==============================
...
collected 3 items

test_count.py
SETUP    S db
        SETUP    F items_db (fixtures used: db)
        test_count.py::test_empty (fixtures used: db, items_db).
        TEARDOWN F items_db
        SETUP    F items_db (fixtures used: db)
        test_count.py::test_count (fixtures used: db, items_db).
        TEARDOWN F items_db
        SETUP    F items_db (fixtures used: db)
        test_count.py::test_count2 (fixtures used: db, items_db).
        TEARDOWN F items_db
TEARDOWN S db
```

(continues on next page)

(continued from previous page)

```
============================== 3 passed in 0.00s ===============================
```

However, if we use the new option, we get a `db` fixture in the `function` scope:

```
$ pytest --fdb --setup-show test_count.py
============================== test session starts ==============================
...
collected 3 items

test_count.py
        SETUP    F db
        SETUP    F items_db (fixtures used: db)
        test_count.py::test_empty (fixtures used: db, items_db).
        TEARDOWN F items_db
        TEARDOWN F db
        SETUP    F db
        SETUP    F items_db (fixtures used: db)
        test_count.py::test_count (fixtures used: db, items_db).
        TEARDOWN F items_db
        TEARDOWN F db
        SETUP    F db
        SETUP    F items_db (fixtures used: db)
        test_count.py::test_count2 (fixtures used: db, items_db).
        TEARDOWN F items_db
        TEARDOWN F db

============================== 3 passed in 0.00s ===============================
```

The database is now set up before each test function and then dismantled again.

`autouse` for fixtures that are always used

Previously, all fixtures used by tests were named by the tests or another fixture in a parameter list. However, you can use `autouse=True` to always run a fixture. This is good for code that needs to run at specific times, but tests are not really dependent on a system state or data from the fixture, for example:

```
import os


@pytest.fixture(autouse=True, scope="session")
def setup_test_env():
    found = os.environ.get("APP_ENV", "")
    os.environ["APP_ENV"] = "TESTING"
    yield
    os.environ["APP_ENV"] = found
```

```
pytest --setup-show test_count.py
============================== test session starts ==============================
...
collected 3 items
```

(continues on next page)

(continued from previous page)

```
test_count.py
SETUP    S setup_test_env
SETUP    S db
        SETUP    F items_db (fixtures used: db)
        test_count.py::test_empty (fixtures used: db, items_db, setup_test_env).
        TEARDOWN F items_db
        SETUP    F items_db (fixtures used: db)
        test_count.py::test_count (fixtures used: db, items_db, setup_test_env).
        TEARDOWN F items_db
        SETUP    F items_db (fixtures used: db)
        test_count.py::test_count2 (fixtures used: db, items_db, setup_test_env).
        TEARDOWN F items_db
TEARDOWN S db
TEARDOWN S setup_test_env

============================== 3 passed in 0.00s ===============================
```

Tip: The `autouse` feature should be the exception rather than the rule. Opt for named fixtures unless you have a really good reason not to do so.

Rename fixtures

The name of a fixture listed in the parameter list of tests and other fixtures that use this fixture is normally the same as the function name of the fixture. However, Pytest allows you to rename fixtures with the `name` parameter to `@pytest.fixture()`:

```
import pytest


from items import cli
@pytest.fixture(scope="session", name="db")
def _db():
    """The db object"""
    yield db()


def test_empty(db):
    assert items_db.count() == 0
```

One case in which renaming can be useful is if the most obvious fixture name already exists as a variable or function name.

Built-in fixtures

Reusing common fixtures is such a good idea that pytest has built in some commonly used fixtures. The built-in fixtures help you to do some very useful things in your tests easily and consistently. Among other things, pytest includes built-in fixtures that can handle temporary directories and files, access command line options, communicate between test sessions, validate output streams, change environment variables and query warnings.

`tmp_path` and `tmp_path_factory`

The tmp_path and tmp_path_factory fixtures are used to create temporary directories. The `tmp_path` fixture for the `function` scope returns a `pathlib.path` instance that points to a temporary directory that persists during the test and a little longer. The `tmp_path_factory` for a session scope fixture returns a `TempPathFactory` object. This object has an `mktemp()` function that returns path objects. With `mktemp()` you can create multiple temporary directories.

In *Test fixtures* we have used the standard library `tempfile.TemporaryDirectory` for our `db` fixture:

```
from pathlib import Path
from tempfile import TemporaryDirectory


@pytest.fixture(scope="session")
def db():
    """ItemsDB object connected to a temporary database"""
    with TemporaryDirectory() as db_dir:
        db_path = Path(db_dir)
        db_ = items.ItemsDB(db_path)
        yield db_
        db_.close()
```

Let's use one of the new built-ins instead. Since our `db` fixture is in the `session` scope, we can't use `tmp_path` because `session` scope fixtures can't use `function` scope fixtures. However, we can use `tmp_path_factory`:

```
@pytest.fixture(scope="session")
def db(tmp_path_factory):
    """ItemsDB object connected to a temporary database"""
    db_path = tmp_path_factory.mktemp("items_db")
    db_ = items.ItemsDB(db_path)
    yield db_
    db_.close()
```

Note: We can also remove two import statements because we don't need to import `pathlib` or `tempfile`.

The base directory for all temporary pytest directories is system and application-dependent. It contains a `pytest-`*`NUM`* part, where *`NUM`* is incremented for each session. The base directory is left unchanged immediately after a session so that you can examine it in the case of test errors. pytest finally cleans them up. Only the last few temporary base directories are left on the system.

You can also specify your own base directory with `pytest --basetemp=`*`MYDIR`*.

capsys

Sometimes the application code should output something to `stdout`, `stderr` ETC. The Items example project therefore also has a command line interface, which we now want to test.

The `items version` command should output the version:

```
$ items version
0.1.0
```

The version is also available via Python:

```
>>> import items
>>> items.__version__
'0.1.0'
```

One way to test this is

1. execute the command with `subprocess.run()`
2. capture the output
3. compare it with the version from the API

```
import subprocess

import items


def test_version():
    process = subprocess.run(["items", "version"], capture_output=True, text=True)
    output = process.stdout.rstrip()
    assert output == items.__version__
```

The `rstrip()` function is used to remove the line break.

The capsys fixture allows us to capture writes to `stdout` and `stderr`. We can call the method that implements this in the CLI directly and use capsys to read the output:

```
import items


def test_version(capsys):
    items.cli.version()
    output = capsys.readouterr().out.rstrip()
    assert output == items.__version__
```

The `capsys.readouterr()` method returns a `namedtuple` that contains `out` and `err`. We only read the `out` part and then we remove the line break with `rstrip()`.

Another feature of `capsys` is the ability to temporarily disable pytest's normal output capture. pytest normally captures the output of your tests and application code. This includes `print` statements.

```
import items


def test_stdout():
```

(continues on next page)

(continued from previous page)

```
    version = items.__version__
    print("\nitems " + version)
```

However, when we run the test, we do not see any output:

```
$ pytest tests/test_output.py
============================== test session starts ==============================
...
collected 1 item

tests/test_output.py .                                                  [100%]

=============================== 1 passed in 0.00s ===============================
```

pytest captures the entire output. While this helps to keep the command line session clean, there may be times when we want to see the entire output, even if the test passes. For this we can use the `-s` or `--capture=no` option:

```
 $ pytest -s tests/test_output.py
 ============================== test session starts ==============================
 ...
 collected 1 item

 tests/test_output.py
 items 0.1.0
 .

 =============================== 1 passed in 0.00s ===============================
```

Another way to always include the output is `capsys.disabled()`:

```
import items


def test_stdout(capsys):
    with capsys.disabled():
        version = items.__version__
        print("\nitems " + version)
```

Now the output is always displayed in the `with` block, even without the `-s` option:

```
$ pytest tests/test_output.py
============================== test session starts ==============================
...
collected 1 item

tests/test_output.py
items 0.1.0
.                                                       [100%]

=============================== 1 passed in 0.00s ===============================
```

See also:

capfd

Like `capsys`, but captures file descriptors 1 and 2, which are normally the same as `stdout` and `stderr`

capsysbinary
While capsys captures text, capsysbinary captures bytes

capfdbinary
captures bytes in file descriptors 1 and 2

caplog
captures output written with the logging package

monkeypatch

With `capsys` I can control the `stdout` and `stderr` output just fine, but it's still not the way I want to test the CLI. The Items application uses a library called Typer, which contains a runner function to test our code the way we would expect a command line test to, which stays in process and provides us with output hooks, for example:

```
from typer.testing import CliRunner

import items


def test_version():
    runner = CliRunner()
    result = runner.invoke(items.app, ["version"])
    output = result.output.rstrip()
    assert output == items.__version__
```

We will use this method of output testing as a starting point for the rest of the Items CLI tests. I started with the CLI tests by testing the Items version. To test the rest of the CLI, we need to redirect the database to a temporary directory, just like we did when testing the API using *fixtures for setup and teardown*. We now use monkeypatch for this:

A monkey patch is a dynamic change to a class or module during runtime. During testing, monkey patching is a convenient way to take over part of the runtime environment of the application code and replace either input or output dependencies with objects or functions that are more suitable for testing. With the built-in fixture `monkeypatch` you can do this in the context of a single test. It is used to change objects, dicts, environment variables, `PYTHONPATH` or the current directory. It's like a mini version of mock. And when the test ends, regardless of whether it passes or fails, the original, unpatched code is restored and everything that was changed by the patch is undone.

See also:

How to monkeypatch/mock modules and environments

The `monkeypatch` fixture offers the following functions:

Function	Description
`setattr(TARGET, NAME, VALUE, raising=True)`[1]	sets an attribute
`delattr(TARGET, NAME, raising=True)`[Page 189, 1]	deletes an attribute
`setitem(DICT, NAME, VALUE)`	sets a dict entry
`delitem(DICT, NAME, raising=True)`[Page 189, 1]	deletes a dict entry
`setenv(NAME, VALUE, prepend=None)`[2]	sets an environment variable
`delenv(NAME, raising=True)`[Page 189, 1]	deletes an environment variable
`syspath_prepend(PATH)`	expands the path `sys.path`
`chdir(PATH)`	changes the current working directory

We can use `monkeypatch` to redirect the CLI to a temporary directory for the database in two ways. Both methods require knowledge of the application code. Let's take a look at the method `cli.get_path()` in `src/items/cli.py`:

```
import os
import pathlib


def get_path():
    db_path_env = os.getenv("ITEMS_DB_DIR", "")
    if db_path_env:
        db_path = pathlib.Path(db_path_env)
    else:
        db_path = pathlib.Path.home() / "items_db"
    return db_path
```

This method tells the rest of the CLI code where the database is located. To display the location of the database on the command line, we now also define `config()` in `src/items/cli.py`:

```
@app.command()
def config():
    """Return the path to the Items db."""
    with items_db() as db:
        print(db.path())
```

```
$ items config
/Users/veit/items_db
```

To test these methods, we can now patch either the entire `get_path()` function or the `pathlib.Path()` attribute `home`. To do this, we first define an auxiliary function `run_items_cli` in `tests/test_config.py`, which outputs the same as `items` on the command line:

```
from typer.testing import CliRunner

import items


def run_items_cli(*params):
    runner = CliRunner()
    result = runner.invoke(items.app, params)
    return result.output.rstrip()
```

We can then write our test, which patches the entire `get_path()` function:

```
def test_get_path(monkeypatch, tmp_path):
    def fake_get_path():
        return tmp_path

    monkeypatch.setattr(items.cli, "get_path", fake_get_path)
    assert run_items_cli("config") == str(tmp_path)
```

The `get_path()` function from `items.cli` cannot simply be replaced by `tmp_path`, as this is a `pathlib.Path`

[1] The `raising` parameter tells pytest whether an exception should be thrown if the element is not (yet) present.

[2] The `prepend` parameter of `setenv()` can be a character. If it is set, the value of the environment variable is changed to *VALUE* + `prepend` + *OLD_VALUE*

object that cannot be called. It is therefore replaced by the `fake_get_path()` function. Alternatively, however, we can also patch the home attribute of `pathlib.Path`:

```
def test_home(monkeypatch, tmp_path):
    items_dir = tmp_path / "items_db"

    def fake_home():
        return tmp_path

    monkeypatch.setattr(items.cli.pathlib.Path, "home", fake_home)
    assert run_items_cli("config") == str(items_dir)
```

However, *monkey patching* and *mocking* complicate testing, so we will look for ways to avoid this whenever possible. In our case, it might be useful to set an environment variable **:envar:`ITEMS_DB_DIR`** that can be easily patched:

```
def test_env_var(monkeypatch, tmp_path):
    monkeypatch.setenv("ITEMS_DB_DIR", str(tmp_path))
    assert run_items_cli("config") == str(tmp_path)
```

Remaining built-in fixtures

Built-in fixture	Description
`capfd`, `capfdbinary`, `capsysbinary`	Variants of `capsys` that work with file descriptors and/or binary output.
`caplog`	similar to `capsys`; used for messages created with Python's logging system.
`cache`	is used to store and retrieve values across multiple Pytest runs. It allows `last-failed`, `failed-first` and similar options.
`doctest_namespace`	useful if you want to use pytest to perform *doctests*.
`pytestconfig`	is used to get access to configuration values, plugin managers and hooks.
`record_property`, `record_testsuite_prop`	is used to add additional properties to the test or test suite. Especially useful for adding data to a report used by CI (Continuous Integration) tools.
`recwarn`	is used to test warning messages.
`request`	is used to provide information about the executed test function. is mostly used in the parameterisation of fixtures.
`pytester`, `testdir`	Used to provide a temporary test directory to support the execution and testing of pytest plugins. `pytester` is the `pathlib` based replacement for the `py.path` based `testdir`.
`tmpdir`, `tmpdir_factory`	similar to `tmp_path` and `tmp_path_factory`; used to return a `py.path.local` object instead of a `pathlib.Path` object.

You can get the complete list of built-in fixtures by running `pytest --fixtures`.

See also:

- Built-in fixtures

Test parameterisation

Parameterisation allows us to convert a test function into many test cases in order to test more thoroughly with less work. To do this, we pass multiple sets of arguments to the test to create new test cases. We'll take a look at redundant code that we can avoid with parameterisation. Then we'll look at three options, in the order in which they should be chosen:

- Parameterisation of functions
- Parameterisation of fixtures
- Using a hook function called `pytest_generate_tests`

We will solve the same parameterisation problem with all three methods, even if sometimes one solution is preferable to the other.

Testing without `parametrize`

Sending some values through a function and checking the output for correctness is a common pattern when testing software. However, calling a function once with a set of values is rarely sufficient to fully test the functions. Parameterised testing is a way to send multiple data sets through the same test and have pytest report if any of the data sets fail. To understand the problem that parameterised tests are trying to solve, let's write some tests for the `finish()` API method from `src/items/api.py`:

```
def finish(self, item_id: int):
    """Set an item state to done."""
    self.update_item(item_id, Item(state="done"))
```

The states used in the application are *todo*, *in progress* and *done*, and `finish()` sets the state of a card to *done*. To test this, we could

1. create an Item object and add it to the database so we have a card to work with
2. call `finish()`
3. ensure that the final state is *done*.

One variable is the start state of the item. It could be "todo", "in progress" or even already "done". Let's test all three:

```
from items import Item


def test_finish_from_in_prog(items_db):
    index = items_db.add_item(Item("Update pytest section", state="in progress"))
    items_db.finish(index)
    item = items_db.get_item(index)
    assert item.state == "done"


def test_finish_from_done(items_db):
    index = items_db.add_item(Item("Update cibuildwheel section", state="done"))
    items_db.finish(index)
    item = items_db.get_item(index)
    assert item.state == "done"


def test_finish_from_todo(items_db):
```

(continues on next page)

(continued from previous page)

```
    index = items_db.add_item(Item("Update mock tests", state="todo"))
    items_db.finish(index)
    item = items_db.get_item(index)
    assert item.state == "done"
```

Let's let it go:

```
pytest -v tests/test_finish.py
============================== test session starts ==============================
...
collected 3 items

tests/test_finish.py::test_finish_from_in_prog PASSED                     [ 33%]
tests/test_finish.py::test_finish_from_done PASSED                        [ 66%]
tests/test_finish.py::test_finish_from_todo PASSED                        [100%]

============================== 3 passed in 0.00s ===============================
```

The test functions are very similar. The only differences are the initial state and the summary. One way to reduce the redundant code is to combine the three functions into a single function, like this:

```
from items import Item


def test_finish(items_db):
    for i in [
        Item("Update pytest section", state="done"),
        Item("Update cibuildwheel section", state="in progress"),
        Item("Update mock tests", state="todo"),
    ]:
        index = items_db.add_item(i)
        items_db.finish(index)
        item = items_db.get_item(index)
        assert item.state == "done"
```

Now we run `tests/test_finish.py` again:

```
$ pytest -v tests/test_finish.py
============================== test session starts ==============================
...
collected 1 item

tests/test_finish.py::test_finish PASSED                                  [100%]

============================== 1 passed in 0.00s ===============================
```

This test has also been passed and we have eliminated the superfluous code. But it's not the same:

- Only one test case is reported instead of three.
- If one of the test cases fails, we don't know which one it is without looking at the traceback or other debugging information.
- If one of the test cases fails, the subsequent test cases are not executed. pytest stops the execution of a test if an assertion fails.

Parameterising functions

To parameterise a test function, add parameters to the test definition and use the `@pytest.mark.parametrize()` decorator to define the arguments to be passed to the test, like this:

```
import pytest

from items import Item


@pytest.mark.parametrize(
    "start_summary, start_state",
    [
        ("Update pytest section", "done"),
        ("Update cibuildwheel section", "in progress"),
        ("Update mock tests", "todo"),
    ],
)
def test_finish(items_db, start_summary, start_state):
    initial_item = Item(summary=start_summary, state=start_state)
    index = items_db.add_item(initial_item)
    items_db.finish(index)
    item = items_db.get_item(index)
    assert item.state == "done"
```

The `test_finish()` function now has its original `items_db` fixture as a parameter, but also two new parameters: `start_summary` and `start_state`. These directly match the first argument of `@pytest.mark.parametrize()`.

1. The first argument of `@pytest.mark.parametrize()` is a list of parameter names. This argument could also be a list of strings, such as `["start_summary", "start_state"]` or a comma-separated string `"start_summary, start_state"`.
2. The second argument of `@pytest.mark.parametrize()` is our list of test cases. Each element in the list is a test case represented by a tuple or list containing one element for each argument sent to the test function.

pytest performs this test once for each `(start_summary, start_state)` pair and reports each as a separate test:

```
$ pytest -v tests/test_finish.py
============================= test session starts ==============================
...
collected 3 items

tests/test_finish.py::test_finish[Update pytest section-done] PASSED     [ 33%]
tests/test_finish.py::test_finish[Update cibuildwheel section-in progress] PASSED [ 66%]
tests/test_finish.py::test_finish[Update mock tests-todo] PASSED         [100%]

============================== 3 passed in 0.00s ===============================
```

This use of `parametrize()` works for our purposes. However, it is not really important for this `test start_summary` and makes every test case more complex. Let's change the parameterisation in `start_state` and see how the syntax changes:

```
import pytest

from items import Item
```

(continues on next page)

(continued from previous page)

```
@pytest.mark.parametrize(
    "start_state",
    [
        "done",
        "in progress",
        "todo",
    ],
)
def test_finish(items_db, start_state):
    i = Item("Update pytest section", state=start_state)
    index = items_db.add_item(i)
    items_db.finish(index)
    item = items_db.get_item(index)
    assert item.state == "done"
```

When we run the tests now, they focus on the change that is important to us:

```
$ pytest -v tests/test_finish.py
============================== test session starts ==============================
...
collected 3 items

tests/test_finish.py::test_finish[done] PASSED                           [ 33%]
tests/test_finish.py::test_finish[in progress] PASSED                    [ 66%]
tests/test_finish.py::test_finish[todo] PASSED                           [100%]

=============================== 3 passed in 0.01s ===============================
```

The output of the two examples differs in that now only the initial state is listed, namely *todo*, *in progress* and *done*. In the previous example, pytest still displayed the values of both parameters, separated by a hyphen -. If only one parameter changes, no hyphen is required.

Parameterising fixtures

During function parameterisation, pytest called our test function once for each set of arguments that we specified. With fixture parameterisation, we move these parameters into a fixture. pytest then calls the fixture once for each set of values we specify. Subsequently, each test function that depends on the fixture is called once for each fixture value. The syntax is also different:

```
import pytest

from items import Item


@pytest.fixture(params=["done", "in progress", "todo"])
def start_state(request):
    return request.param
```

(continues on next page)

(continued from previous page)

```
def test_finish(items_db, start_state):
    i = Item("Update pytest section", state=start_state)
    index = items_db.add_item(i)
    items_db.finish(index)
    item = items_db.get_item(index)
    assert item.state == "done"
```

This means that pytest calls `start_state()` three times, once for each of the values in `params`. Each value of `params` is stored in `request.param` so that the fixture can use it. Within `start_state()` we could have code that depends on the parameter value. In this case, however, only the value of the parameter is returned.

The function `test_finish()` is identical to the function we used in the function parameterisation, but without the decorator `parametrize`. Since it has `start_state` as a parameter, pytest calls it once for each value that is passed to the `start_state()` fixture. And after all this, the output looks exactly the same as before:

```
$ pytest -v tests/test_finish.py
============================== test session starts ==============================
...
collected 3 items

tests/test_finish.py::test_finish[done] PASSED                           [ 33%]
tests/test_finish.py::test_finish[in progress] PASSED                    [ 66%]
tests/test_finish.py::test_finish[todo] PASSED                           [100%]

=============================== 3 passed in 0.01s ===============================
```

At first glance, fixture parameterisation fulfils roughly the same purpose as function parameterisation, but with a little more code. However, fixture parameterisation has the advantage that a fixture is executed for each set of arguments. This is useful if you have setup or teardown code that needs to be executed for each test case, for example a different database connection or file content or whatever.

It also has the advantage that many test functions can be executed with the same set of parameters. All tests that use the `start_state` fixture are called all three times, once for each `start state`.

Parameterise with `pytest_generate_tests`

The third option for parameterisation is to use a hook function called `pytest_generate_tests`. Hook functions are often used by *Plugins* to change the normal workflow of pytest. But we can use many of them in test files and `conftest.py` files.

The implementation of the same flow as before with `pytest_generate_tests` looks like this:

```
from items import Item


def pytest_generate_tests(metafunc):
    if "start_state" in metafunc.fixturenames:
        metafunc.parametrize("start_state", ["done", "in progress", "todo"])


def test_finish(items_db, start_state):
    i = Item("Update pytest section", state=start_state)
    index = items_db.add_item(i)
```

(continues on next page)

(continued from previous page)

```
    items_db.finish(index)
    item = items_db.get_item(index)
    assert item.state == "done"
```

The `test_finish()` function has not changed; we have only changed the way pytest enters the value for `initial_state` for each test call.

The `pytest_generate_tests` function that we provide is called by pytest when it generates its list of tests to run. It is very powerful and our example is just a simple case of matching the functionality of previous parameterisation methods. However, `pytest_generate_tests` is particularly useful if we want to change the parameterisation list at test collection time in an interesting way. Here are a few possibilities:

- We could change our parameterisation list based on a command line option that `metafunc.config.getoption("--SOME_OPTION")`[1] gives us. Maybe we add an `--excessive` option to test more values, or a `--quick` option to test only a few.
- The parameterisation list of a parameter can be based on the presence of another parameter. For example, for test functions that query two related parameters, we can parameterise both with a different set of values than if the test queries only one of the parameters.
- We can parameterise two related parameters at the same time, for example `metafunc.parametrize("TUTORIAL, TOPIC", [("PYTHON BASICS", "TESTING"), ("PYTHON BASICS", "DOCUMENTING"), ("PYTHON FOR DATA SCIENCE, "GIT"), ...])`.

We have now become familiar with three ways of parameterising tests. Although we only create three test cases from one test function in the *finish()* example, parameterisation can generate a large number of test cases.

Markers

Markers in pytest can be thought of as tags or labels. If some tests are slow, you can mark them with `@pytest.mark.slow` and have pytest skip those tests if you are in a hurry. You can select a handful of tests from a test suite and mark them with `@pytest.mark.smoke` and run them as the first stage of a test pipeline in a *CI* system. You can really use markers for any reason you have to run just a few tests.

pytest contains a handful of built-in markers that change the behaviour of the test execution. We have already used one of these, `@pytest.mark.parametrize`, in *Parameterising functions*. In addition to the custom markers we can create and add to our tests, the built-in markers tell pytest to do something special with the marked tests.

Below, we will explore both types of markers in more detail: the built-in markers that change behaviour and the custom markers that we can create to select which tests to run. We can also use markers to pass information to a fixture that is used by a test.

Using built-in markers

The pytest built-in markers are used to modify the test execution. Here is the complete list of built-in markers included in pytest:

`@pytest.mark.filterwarnings(WARNUNG)`
: This marker adds a warning filter to the specified test.

`@pytest.mark.skip(reason=None)`
: This marker skips the test with an optional reason.

[1] https://docs.pytest.org/en/latest/reference.html#metafunc

@pytest.mark.skipif(*BEDINGUNG*, ...*, *GRUND*)
: This marker skips the test if one of the conditions is `True`.

@pytest.mark.xfail(*BEDINGUNG*, ...* *GRUND*, run=*True*, raises=*None*, strict=*xfail_strict*)
: This marker tells pytest that we expect the test to fail.

@pytest.mark.parametrize({ARG1, ARG2, ...
: This marker calls a test function several times, passing different arguments one after the other.

@pytest.mark.usefixtures({FIXTURE1, FIXTURE2, ...
: This marker identifies tests that require all the specified fixtures.

We have already used *@pytest.mark.parametrize*. Let's go through the other three most commonly used built-in markers with some examples to see how they work.

Skipping tests with `@pytest.mark.skip`

The `skip` marker allows us to skip a test. Let's say we want to add the ability to sort in a future version of the `Items` application and want the `Item` class to support comparisons. We write a test for comparing `Item` objects with < as follows:

```
from items import Item


def test_less_than():
    i1 = Item("Update pytest section")
    i2 = Item("Update cibuildwheel section")
    assert i1 < i2


def test_equality():
    i1 = Item("Update pytest section")
    i2 = Item("Update pytest section")
    assert i1 == i2
```

And it fails:

```
pytest --tb=short tests/test_compare.py
============================= test session starts ==============================
...
collected 2 items

tests/test_compare.py F.                                                 [100%]

=================================== FAILURES ===================================
_____________________________ test_less_than ______________________________
tests/test_compare.py:7: in test_less_than
    assert i1 < i2
E   TypeError: '<' not supported between instances of 'Item' and 'Item'
=========================== short test summary info ============================
FAILED tests/test_compare.py::test_less_than - TypeError: '<' not supported between␣
↪instances of 'Item' and 'Item'
========================= 1 failed, 1 passed in 0.03s ==========================
```

The error is simply due to the fact that we have not yet implemented this function. However, we don't have to throw this test away again; we can simply omit it:

```
import pytest

from items import Item


@pytest.mark.skip(reason="Items do not yet allow a < comparison")
def test_less_than():
    i1 = Item("Update pytest section")
    i2 = Item("Update cibuildwheel section")
    assert i1 < i2
```

The marker `@pytest.mark.skip()` instructs pytest to skip the test. Specifying a reason is optional, but it helps with further development. When we execute skipped tests, they are displayed as `s`:

```
$ pytest --tb=short tests/test_compare.py
============================= test session starts ==============================
...
collected 2 items

tests/test_compare.py s.                                                 [100%]

========================= 1 passed, 1 skipped in 0.00s =========================
```

... or verbos as `SKIPPED`:

```
$ pytest -v -ra tests/test_compare.py
============================= test session starts ==============================
...
collected 2 items

tests/test_compare.py::test_less_than SKIPPED (Items do not yet allo...) [ 50%]
tests/test_compare.py::test_equality PASSED                              [100%]

============================= short test summary info ==========================
SKIPPED [1] tests/test_compare.py:6: Items do not yet allow a < comparison
========================= 1 passed, 1 skipped in 0.00s =========================
```

Since we have instructed pytest with `-r` to output a short summary of our tests, we get an additional line at the bottom that lists the reason we specified in the marker. The `a` in `-ra` stands for *all except passed*. The `-ra` options are the most common, as we almost always want to know why certain tests failed.

See also:

- Skipping test functions

Conditional skipping of tests with `@pytest.mark.skipif`

Suppose we know that we will not support sorting in versions 0.1.x of the Items app, but we will support it in version 0.2.x. Then we can instruct pytest to skip the test for all versions of items lower than 0.2.x as follows:

```
import pytest
from packaging.version import parse

import items
from items import Item


@pytest.mark.skipif(
parse(items.__version__).minor < 2,
reason="The comparison with < is not yet supported in version 0.1.x.",
)
def test_less_than():
    i1 = Item("Update pytest section")
    i2 = Item("Update cibuildwheel section")
    assert i1 < i2
```

With the `skipif` marker, you can enter as many conditions as you like, and if one of them is true, the test is skipped. In our case, we use `packaging.version.parse` to isolate the minor version and compare it with the number 2.

In this example, packaging is used as an additional package. If you want to try out the example, install it first with `python -m pip install packaging`.

Tip: `skipif` is also ideal if tests need to be written differently for different operating systems.

See also:

- skipif

`@pytest.mark.xfail`

If we want to run all tests, even those that we know will fail, we can use the marker `xfail` or more precisely `@pytest.mark.xfail(CONDITION, ... *, {REASON, run=True, raises=None, strict=True)`. The first set of parameters for this fixture is the same as for `skipif`.

`run`
: The test is executed by default, unless `run=False` is set.

`raises`
: allows you to specify an exception type or a tuple of exception types that should result in an `xfail`. Any other exception will cause the test to fail.

`strict`
: tells pytest whether passed tests `(strict=False)` should be marked as `XPASS` or with `strict=True` as `FAIL`.

Let's take a look at an example:

```
import pytest
from packaging.version import parse

```

(continues on next page)

(continued from previous page)

```
import items
from items import Item


@pytest.mark.xfail(
parse(items.__version__).minor < 2,
reason="The comparison with < is not yet supported in version 0.1.x.",
)
def test_less_than():
    i1 = Item("Update pytest section")
    i2 = Item("Update cibuildwheel section")
    assert i1 < i2

@pytest.mark.xfail(reason="Feature #17: not implemented yet")
def test_xpass():
    i1 = Item("Update pytest section")
    i2 = Item("Update pytest section")
    assert i1 == i2

@pytest.mark.xfail(reason="Feature #17: not implemented yet", strict=True)
def test_xfail_strict():
    i1 = Item("Update pytest section")
    i2 = Item("Update pytest section")
    assert i1 == i2
```

We have three tests here: one that we know will fail, and two that we know will pass. These tests demonstrate both the failing and passing of using `xfail` and the effects of using `strict`. The first example also uses the optional `condition` parameter, which works like `skipif`'s conditions. And this is what the result looks like:

```
pytest -v -ra tests/test_xfail.py
============================= test session starts ==============================
...
collected 3 items

tests/test_xfail.py::test_less_than XFAIL (The comparison with < is ...) [ 33%]
tests/test_xfail.py::test_xpass XPASS (Feature #17: not implemented yet) [ 66%]
tests/test_xfail.py::test_xfail_strict FAILED                            [100%]

=================================== FAILURES ===================================
_____________________________ test_xfail_strict _____________________________
[XPASS(strict)] Feature #17: not implemented yet
=========================== short test summary info ============================
XFAIL tests/test_xfail.py::test_less_than - The comparison with < is not yet supported␣
↪in version 0.1.x.
XPASS tests/test_xfail.py::test_xpass Feature #17: not implemented yet
FAILED tests/test_xfail.py::test_xfail_strict
==================== 1 failed, 1 xfailed, 1 xpassed in 0.02s ====================
```

Tests labelled with `xfail`:

- Failed tests are displayed with `XFAIL`.
- Passed tests with `strict=False` result in `XPASSED`.

- Passed tests with `strict=True` result in `FAILED`.

If a test fails that is marked with `xfail`, which means it is output with `XFAIL`, we were right in assuming that the test will fail.

For tests that were marked `xfail` but actually passed, there are two possibilities: If they are supposed to result in `XFAIL`, then you should keep your hands off strictly. If, on the other hand, they should result in `FAILED`, then set `strict`. You can either set `strict` as an option for the `xfail` marker, as we have done in this example, or you can also set it globally with the setting `xfail_strict=True` in the pytest configuration file `pytest.ini`.

A pragmatic reason to always use `xfail_strict=True` is that we usually take a closer look at all failed tests. And so we also look at the cases in which the expectations of the test do not match the result.

`xfail` can be very helpful if you are working in test-driven development and you are writing test cases that you know are not yet implemented but that you want to implement soon. Leave the `xfail` tests on the feature branch in which the function is implemented.

Or something breaks, one or more tests fail, and you can't work on fixing it right away. Marking the tests as `xfail`, `strict=true` with the error/issue report ID in reason is a good way to keep the test running and not forget about it.

However, if you are just brainstorming about the behaviours of your application, you should not write tests and mark them with `xfail` or `skip` yet: here I would recommend YAGNI ('You Aren't Gonna Need It'). Always implement things only when they are actually needed and never when you only suspect that you will need them.

Tip:

- You should set `xfail_strict = True` in `pytest.ini` to turn all `XPASSED` results into `FAILED`.
- You should also always use `-ra` or at least `-rxX` to display the reason.
- And finally, you should specify an error number in `reason`.
- `pytest --runxfail` basically ignores the `xfail` markers. This is very useful in the final stages of pre-production testing.

Selection of tests with your own markers

You can think of your own markers as tags or labels. They can be used to select tests that should be executed or skipped.

Let's say we want to label some of our tests with `smoke`. Segmenting a subset of tests into a smoke test suite is a common practice to be able to run a representative set of tests that can quickly tell us if anything is wrong with any of the main systems. In addition, we will label some of our tests with `exception` – those that check for expected exceptions:

```
import pytest

from items import InvalidItemId, Item


@pytest.mark.smoke
def test_start(items_db):
    """
    Change state from 'todo' to 'in progress'
    """
    i = items_db.add_item(Item("Update pytest section", state="todo"))
    items_db.start(i)
```

(continues on next page)

(continued from previous page)

```
    s = items_db.get_item(i)
    assert s.state == "in progress"
```

Now we should be able to select only this test by using the `-m smoke` option:

```
$ pytest -v -m smoke tests/test_start.py
============================== test session starts ==============================
...
collected 2 items / 1 deselected / 1 selected

tests/test_start.py::test_start PASSED                                   [100%]

=============================== warnings summary ================================
tests/test_start.py:6
  /Users/veit/items/tests/test_start.py:6: PytestUnknownMarkWarning: Unknown pytest.mark.
↪smoke - is this a typo?  You can register custom marks to avoid this warning - for␣
↪details, see https://docs.pytest.org/en/stable/how-to/mark.html
    @pytest.mark.smoke

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
================== 1 passed, 1 deselected, 1 warning in 0.00s ==================
```

Now we were only able to run one test, but we also received a warning: `PytestUnknownMarkWarning: Unknown pytest.mark.smoke - is this a typo?` It helps to avoid typos. pytest wants us to register custom markers by adding a marker section to `pytest.ini`, for example:

```
[pytest]
markers =
    smoke: Small subset of all tests
```

Now pytest no longer warns us of an unknown marker:

```
$ pytest -v -m smoke tests/test_start.py
============================== test session starts ==============================
...
configfile: pytest.ini
collected 2 items / 1 deselected / 1 selected

tests/test_start.py::test_start PASSED                                   [100%]

======================= 1 passed, 1 deselected in 0.00s ========================
```

Let's do the same with the `exception` marker for `test_start_non_existent`.

1. First, we register the marker in `pytest.ini`:

   ```
   [pytest]
   markers =
       smoke: Small subset of tests
       exception: Only run expected exceptions
   ```

2. Then we add the marker to the test:

```
@pytest.mark.exception
def test_start_non_existent(items_db):
    """
    Shouldn't start a non-existent item.
    """
    # any_number will be invalid, db is empty
    any_number = 44

    with pytest.raises(InvalidItemId):
        items_db.start(any_number)
```

3. Finally, we run the test with `-m exception`:

```
$ pytest -v -m exception tests/test_start.py
============================= test session starts ==============================
...
configfile: pytest.ini
collected 2 items / 1 deselected / 1 selected

tests/test_start.py::test_start_non_existent PASSED                      [100%]

======================= 1 passed, 1 deselected in 0.01s ========================
```

Markers for files, classes and parameters

With the tests in `test_start.py`, we have added `@pytest.mark.`*`MARKER_NAME`* decorators to test functions. We can also add markers to entire files or classes to mark multiple tests, or go into parameterised tests and mark individual parameterisations. We can even set multiple markers on a single test. First, we set in `test_finish.py` with a file-level marker:

```
import pytest

from items import Item

pytestmark = pytest.mark.finish
```

If pytest sees a `pytestmark` attribute in a test module, it will apply the marker(s) to all tests in that module. If you want to apply more than one marker to the file, you can use a list form: `pytestmark = [pytest.mark.`*`MARKER_ONE`*`, pytest.mark.`*`MARKER_TWO`*`]`.

Another way to mark multiple tests at the same time is to have tests in a class and use markers at class level:

```
@pytest.mark.smoke
class TestFinish:
    def test_finish_from_todo(self, items_db):
        i = items_db.add_item(Item("Update pytest section", state="todo"))
        items_db.finish(i)
        s = items_db.get_item(i)
        assert s.state == "done"

    def test_finish_from_in_prog(self, items_db):
        i = items_db.add_item(Item("Update pytest section", state="in progress"))
```

(continues on next page)

(continued from previous page)

```
        items_db.finish(i)
        s = items_db.get_item(i)
        assert s.state == "done"

    def test_finish_from_done(self, items_db):
        i = items_db.add_item(Item("Update pytest section", state="done"))
        items_db.finish(i)
        s = items_db.get_item(i)
        assert s.state == "done"
```

The test class `TestFinish` is labelled with `@pytest.mark.smoke`. If you mark a test class in this way, every test method in the class will be labelled with the same marker.

We can also mark only certain test cases of a parameterised test:

```
@pytest.mark.parametrize(
    "states",
    [
        "todo",
        pytest.param("in progress", marks=pytest.mark.smoke),
        "done",
    ],
)
def test_finish(items_db, start_state):
    i = items_db.add_item(Item("Update pytest section", state=states))
    items_db.finish(i)
    s = items_db.get_item(i)
    assert s.state == "done"
```

The `test_finish()` function is not directly marked, but only one of its parameters: `pytest.param("in progress", marks=pytest.mark.smoke)`. You can use more than one marker by using the list form: `marks=[pytest.mark.ONE, pytest.mark.TWO]`. If you want to mark all test cases of a parameterised test, insert the marker either above or below the decorator `parametrize`, as with a normal function.

The previous example referred to function parameterisation. However, you can also mark fixtures in the same way:

```
@pytest.fixture(
    params=[
        "todo",
        pytest.param("in progress", marks=pytest.mark.smoke),
        "done",
    ]
)
def start_state_fixture(request):
    return request.param


def test_finish(items_db, start_state_fixture):
    i = items_db.add_item(Item("Update pytest section", state=start_state_fixture))
    items_db.finish(i)
    s = items_db.get_item(i)
    assert s.state == "done"
```

If you want to add more than one marker to a function, you can simply stack them. For example,

`test_finish_non_existent()` is marked with both `@pytest.mark.smoke` and `@pytest.mark.exception`:

```
from items import InvalidItemId, Item


@pytest.mark.smoke
@pytest.mark.exception
def test_finish_non_existent(items_db):
    i = 44 # any_number will be invalid, db is empty
    with pytest.raises(InvalidItemId):
        items_db.finish(i)
```

We have added a number of markers to `test_finish.py` in various ways. We use the markers to select the tests to be executed instead of a test file:

```
$ cd tests
$ tests % pytest -v -m exception
============================= test session starts ==============================
...
configfile: pytest.ini
collected 36 items / 34 deselected / 2 selected

test_finish.py::test_finish_non_existent PASSED                          [ 50%]
test_start.py::test_start_non_existent PASSED                            [100%]

======================= 2 passed, 34 deselected in 0.07s =======================
```

Markers together with `and`, `or`, `not` and `()`

We can logically combine markers to select tests, just like we used `-k` together with keywords to select test cases in a *test suite*. So we can only select the `finish` tests that deal with `exception`:

```
pytest -v -m "finish and exception"
============================= test session starts ==============================
...
configfile: pytest.ini
collected 36 items / 35 deselected / 1 selected

test_finish.py::test_finish_non_existent PASSED                          [100%]

======================= 1 passed, 35 deselected in 0.08s =======================
```

We can also use all logical operations together:

```
 $ pytest -v -m "(exception or smoke) and (not finish)"
============================= test session starts ==============================
...
configfile: pytest.ini
collected 36 items / 34 deselected / 2 selected

test_start.py::test_start PASSED                                         [ 50%]
test_start.py::test_start_non_existent PASSED                            [100%]
```

(continues on next page)

(continued from previous page)

```
====================== 2 passed, 34 deselected in 0.08s ======================
```

Finally, we can also combine markers and keywords for the selection, for example, to perform smoke tests that are not part of the `TestFinish` class:

```
$ pytest -v -m smoke -k "not TestFinish"
============================= test session starts ==============================
...
configfile: pytest.ini
collected 36 items / 33 deselected / 3 selected

test_finish.py::test_finish[in progress] PASSED                          [ 33%]
test_finish.py::test_finish_non_existent PASSED                          [ 66%]
test_start.py::test_start PASSED                                         [100%]

====================== 3 passed, 33 deselected in 0.07s ======================
```

When using markers and keywords, please note that the names of the markers must be complete with the `-m` *`MARKERNAME`* option, while keywords are more of a substring with the `-k` *`KEYWORD`* option.

`--strict-markers`

Usually we get a warning if a marker is not registered. If we want this warning to be an error instead, we can use the `--strict-markers` option. This has two advantages:

1. The error is already output when the tests to be executed are collected and not at runtime. If you have a test suite that takes longer than a few seconds, you will appreciate getting this feedback quickly.
2. Secondly, errors are sometimes easier to recognise than warnings, especially in systems with *continuous integration*.

Tip: It is therefore recommended to always use `--strict-markers`. However, instead of entering the option again and again, you can add `--strict-markers` to the `addopts` section of `pytest.ini`:

```
[pytest]
...
addopts =
    --strict-markers
```

Combining markers with fixtures

Markers can be used in conjunction with fixtures, plugins and hook functions. The built-in markers require parameters, while the custom markers we have used so far do not require parameters. Let's create a new marker called `num_items` that we can pass to the `items_db` fixture. The `items_db` fixture currently cleans up the database for each test that wants to use it:

```
@pytest.fixture(scope="function")
    def items_db(session_items_db):
    db = session_items_db
```

(continues on next page)

(continued from previous page)

```
    db.delete_all()
    return db
```

For example, if we want to have four items in the database when our test starts, we can simply write a different but similar fixture:

```
@pytest.fixture(scope="session")
def items_list():
    """List of different Item objects"""
    return [
        items.Item("Add Python 3.12 static type improvements", "veit", "todo"),
        items.Item("Add tips for efficient testing", "veit", "wip"),
        items.Item("Update cibuildwheel section", "veit", "done"),
        items.Item("Add backend examples", "veit", "done"),
    ]


@pytest.fixture(scope="function")
def populated_db(items_db, items_list):
    """ItemsDB object populated with 'items_list'"""
    for i in items_list:
        items_db.add_item(i)
    return items_db
```

We could then use the original fixture for tests, which provides an empty database, and the new fixture for tests, which contains a database with four items:

```
def test_zero_item(items_db):
    assert items_db.count() == 0


def test_four_items(populated_db):
    assert populated_db.count() == 4
```

We now have the option of testing either zero or four items in the database. But what if we want to have no, four or 13 items? Then we don't want to write a new fixture each time. Markers allow us to tell a test how many items we want to have. This requires three steps:

1. First, we define three different tests in `test_items.py` with our marker `@pytest.mark.num_items`:

   ```
   @pytest.mark.num_items
   def test_zero_item(items_db):
       assert items_db.count() == 0


   @pytest.mark.num_items(4)
   def test_four_items(items_db):
       assert items_db.count() == 4


   @pytest.mark.num_items(13)
   def test_thirteen_items(items_db):
       assert items_db.count() == 13
   ```

2. We must then declare this marker in the `pytest.ini` file:

```
[pytest]
markers =
    ...
    num_items: Number of items to be pre-filled for the items_db fixture
```

3. Now we modify the `items_db` fixture in the `conftest.py` file to be able to use the marker. To avoid having to hard-code the item information, we will use the Python package Faker, which we can install with `python -m pip install faker`:

```
1  import os
2  from pathlib import Path
3  from tempfile import TemporaryDirectory
4
5  import faker
6  import pytest
7
8  import items
9
10 ...
11
12 @pytest.fixture(scope="function")
13 def items_db(session_items_db, request, faker):
14     db = session_items_db
15     db.delete_all()
16     # Support for random selection "@pytest.mark.num_items({NUMBER})`.
17     faker.seed_instance(99)
18     m = request.node.get_closest_marker("num_items")
19     if m and len(m.args) > 0:
20         num_items = m.args[0]
21         for _ in range(num_items):
22             db.add_item(Item(summary=faker.sentence(), owner=faker.first_name()))
23     return db
```

There are a lot of changes here that we want to go through now.

Line 13
We have added `request` and `faker` to the list of `items_db` parameters.

Line 17
This sets the randomness of faker so that we get the same data every time. We are not using faker here for very random data, but to avoid having to invent data ourselves.

Line 18
Here we use `request`, more precisely `request.node` for the pytest representation of a test. `get_closest_marker('num_items')` returns a marker object if the test is marked with `num_items`, otherwise it returns `None`. The `get_closest_marker()` function returns the marker closest to the test, which is usually what we want.

Line 19
The expression is true if the test is marked with `num_items` and an argument is given. The additional `len` check is there so that if someone accidentally just uses `pytest.mark.num_items` without specifying the number of items, this part is skipped.

Line 20–22
Once we know how many items we need to create, we let Faker create some data for us. Faker provides the

Faker fixture.

- For the `summary` field, the `faker.sentence()` method works.
- The `faker.first_name()` method works for the `Owner` field.

See also:

- There are many other options that you can use with Faker. Have a look at the Faker documentation.
- In addition to Faker, there are other libraries that provide fake data, see *Fake plugins*.

Let's run the tests now to make sure everything is working properly:

```
$ pytest -v -s test_items.py
============================= test session starts ==============================
...
configfile: pytest.ini
plugins: Faker-19.10.0
collected 3 items

test_items.py::test_zero_item PASSED
test_items.py::test_four_items PASSED
test_items.py::test_thirteen_items PASSED

============================== 3 passed in 0.09s ===============================
```

Note: You can add a `print` statement to `test_four_items()` to get an impression of what the data from Faker looks like:

```
@pytest.mark.num_items(4)
def test_four_items(items_db):
    assert items_db.count() == 4
    print()
    for i in items_db.list_items():
        print(i)
```

You can then call the tests in `test_items.py` again:

```
$ pytest -v -s test_items.py
============================= test session starts ==============================
...
configfile: pytest.ini
plugins: Faker-19.10.0
collected 3 items

test_items.py::test_zero_item PASSED
test_items.py::test_four_items
Item(summary='Herself outside discover card beautiful rock.', owner='Alyssa', state='todo
↪', id=1)
Item(summary='Bed perhaps current reveal open society small.', owner='Lynn', state='todo
↪', id=2)
Item(summary='Charge produce sure full water.', owner='Allison', state='todo', id=3)
Item(summary='Light I especially account.', owner='James', state='todo', id=4)
PASSED
test_items.py::test_thirteen_items PASSED
```

(continues on next page)

(continued from previous page)

```
============================== 3 passed in 0.09s ===============================
```

List markers

We've already covered a lot of markers: the built-in markers `skip`, `skipif` and `xfail`, our own markers `smoke`, `exception`, `finish` and `num_items` and there are also a few more built-in markers. And when we start using *Plugins*, more markers may be added. To list all available markers with descriptions and parameters, you can run `pytest --markers`:

```
$ pytest --markers
@pytest.mark.exception: Only run expected exceptions

@pytest.mark.finish: Only run finish tests

@pytest.mark.smoke: Small subset of all tests

@pytest.mark.num_items: Number of items to be pre-filled for the items_db fixture

@pytest.mark.filterwarnings(warning): add a warning filter to the given test. see https:/
↪/docs.pytest.org/en/stable/how-to/capture-warnings.html#pytest-mark-filterwarnings
...
```

This is a very handy feature that allows us to quickly search for markers and a good reason to add useful descriptions to our own markers.

Plugins

As powerful as pytest is, it can do even more when we add plugins. The pytest codebase is designed to allow customisation and extensions, and there are hooks that allow changes and improvements through plugins.

You may be surprised to find that you have already written some plugins if you have worked through the previous sections. Every time you add fixtures or hook functions to a project's `conftest.py` file, you are creating a local plugin. It's just a little extra work to turn these `conftest.py` files into installable plugins that you can share between projects, with other people, or with the world.

But first, let's start with where you can find third-party plugins. There are quite a few plugins out there, so there's a good chance that any changes you want to make to pytest have already been written.

Finding plugins

You can find third-party pytest plugins in various places, for example the pytest documentation contains an alphabetical list of plugins from *pypi.org*. You can also search pypi.org itself, for pytest or for the pytest framework. Finally, many popular pytest plugins can also be found in pytest-dev on GitHub.

Installing plugins

Like other Python packages, pytest plugins can be easily installed with *pip*: `python -m pip install pytest-cov`.

Plugins for ...

... modified test sequences

pytest usually executes our tests in a predictable order. For a directory of test files, pytest executes each file in alphabetical order. Within each file, each test is executed in the order in which it appears in the file. However, it can sometimes be useful to change this order. The following plugins change the usual sequence of a test:

pytest-xdist
: executes tests in parallel, either with several CPUs on one machine or several remote machines.

pytest-rerunfailures
: re-executes failed tests and is particularly helpful in the case of faulty tests.

pytest-repeat
: makes it easy to repeat one or more tests.

pytest-order
: enables the order to be defined using *Markers*.

pytest-randomly
: runs the tests in random order, first by file, then by class, then by test file.

... modified output

The normal pytest output mainly shows dots for passed tests and characters for other output. If you pass `-v`, you will see a list of test names with the result. However, there are plugins that change the output even further:

pytest-instafail
: adds a `--instafail` option that reports tracebacks and output from failed tests immediately after the failure. Normally, pytest reports tracebacks and output from failed tests only after all tests have completed.

pytest-sugar
: shows green checkmarks instead of dots for passed tests and has a nice progress bar. Like pytest-instafail, it also shows failures immediately.

pytest-html
: enables the creation of HTML reports. Reports can be extended with additional data and images, such as screenshots of error cases.

pytest-icdiff
: improves diffs in the error messages of the pytest assertion with ICDiff.

... web development

pytest is used extensively for testing web projects and there is a long list of plugins that further simplify testing:

pytest-selenium
: provides fixtures that enable simple configuration of browser-based tests with Selenium.

pytest-splinter
: provide the high-level API of the Selenium-based Splinter to be used more easily from pytest.

pytest-httpx
: facilitates the testing of HTTPX and FastAPI applications.

... fake data

We have already used Faker in *Combining markers with fixtures* to create multiple item instances. There are many cases in different areas where it is helpful to generate fake data. It is therefore not surprising that there are several plugins that fulfil this need:

Faker
: generates fake data for you and offers a faker fixture for use with pytest.

pytest-factoryboy
: contains fixtures for factory-boy, a database model data generator.

pytest-mimesis
: generates fake data similar to Faker, but Mimesis is a lot faster.

... various things

pytest-cov
: executes the *Coverage* during testing.

pytest-benchmark
: performs benchmark timing for code within tests.

pytest-timeout
: prevents tests from running too long.

pytest-asyncio
: tests asynchronous functions.

pytest-mock
: is a thin wrapper around the *unittest.mock* patching API.

pytest-freezegun
: freezes the time so that any code that reads the time, date or clock time will get the same value during a test. set a specific time.

pytest-grpc
: is a Pytest plugin for gRPC.

pytest-bdd
: writes BDD (Behavior Driven Development) tests with pytest.

Own plugins

See also:

- Writing plugins

Configuration

You can use configuration files to change the way pytest runs. If you repeatedly use certain options in your tests, such as `--verbose` or `--strict-markers`, you can store them in a configuration file so that you don't have to enter them again and again. In addition to the configuration files, there are a handful of other files that are helpful when using pytest to make writing and running tests easier:

`pytest.ini`
: This is the most important configuration file of pytest, with which you can change the default behaviour of pytest. It also defines the root directory of pytest, or `rootdir`.

`conftest.py`
: This file contains *Test fixtures* and hook functions. It can exist in `rootdir` or in any subdirectory.

`__init__.py`
: If this file is stored in test subdirectories, it enables the use of identical test file names in several test directories.

If you already have a `tox.ini`, `pyproject.toml` or `setup.cfg` in your project, they can take the place of the `pytest.ini` file: `tox.ini` is used by *tox*, `pyproject.toml` and `setup.cfg` are used for packaging Python projects and can be used to store settings for various tools, including pytest.

You should have a configuration file, either `pytest.ini`, or a `pytest` section in `tox.ini`, `pyproject.toml` or in `setup.cfg`.

The configuration file defines the top-level directory from which `pytest` is started.

Let's take a look at some of these files in the context of a project directory structure:

```
items
├── ...
├── pytest.ini
├── src
│   └── ...
└── tests
    ├── __init__.py
    ├── conftest.py
    └── test_....py
```

In the case of the `items` project that we have used for testing so far, there is a `pytest.ini` file and a `tests` directory at the top level. We will refer to this structure when we talk about the various files in the rest of this section.

Saving settings and options in `pytest.ini`

```
[pytest]
addopts =
    --strict-markers
    --strict-config
    -ra
testpaths = tests
markers =
    smoke: Small subset of all tests
    exception: Only run expected exceptions
```

`[pytest]` marks the start of the pytest section. This is followed by the individual settings. For configuration settings that allow more than one value, the values can be written either in one or more lines in the form *`SETTING = VALUE1 VALUE2`*. With `markers`, however, only one marker per line is permitted.

This example is a simple `pytest.ini` file that I use in almost all my projects. Let's briefly go through the individual lines:

`addopts =`
: allows you to specify the pytest options that we always want to execute in this project.

`--strict-markers`
: instructs pytest to issue an error instead of a warning for every unregistered marker that appears in the test code. This allows us to avoid typos in marker names.

`--strict-config`
: instructs pytest to issue an error instead of a warning if difficulties arise when parsing configuration files. This prevents typing errors in the configuration file from going unnoticed.

`-ra`
: instructs pytest to display not only additional information on failures and errors at the end of a test run, but also a test summary.

  `-r`
  : displays additional information on the test summary.

  `a`
  : displays all but the passed tests. This adds the information `skipped`, `xfailed` or `xpassed` to the failures and errors.

`testpaths = tests`
: tells pytest where to look for tests if you have not specified a file or directory name on the command line. In our case, pytest searches in the `tests` directory.

  At first glance, it may seem superfluous to set `testpaths` to `tests`, as pytest searches there anyway and we do not have any `test_` files in our `src` or `docs` directories. However, specifying a `testpaths` directory can save a little startup time, especially if our `src`, `docs` or other directories are quite large.

`markers =`
: is used to declare markers, as described in *Selection of tests with your own markers*.

See also:

You can specify many other configuration settings and command line options in the configuration files, which you can display using the `pytest --help` command.

Using other configuration files

If you are writing tests for a project that already has a `pyproject.toml`, `tox.ini` or `setup.cfg` file, you can use `pytest.ini` to store your pytest configuration settings, or you can store your configuration settings in one of these alternative configuration files. The syntax of the two non-ini files is slightly different, so we will take a closer look at both files.

pyproject.toml

The `pyproject.toml` file was originally intended for the packaging of Python projects; however, it can also be used to define project settings.

As TOML is a different standard for configuration files than `.ini` files, the format is also slightly different:

```
[tool.pytest.ini_options]
addopts = [
    "--strict-markers",
    "--strict-config",
    "-ra"
    ]
testpaths = "tests"
markers = [
    "exception: Only run expected exceptions",
    "finish: Only run finish tests",
    "smoke: Small subset of all tests",
    "num_items: Number of items to be pre-filled for the items_db fixture"
    ]
```

Instead of `[pytest]`, the section begins with `[tool.pytest.ini_options]`, the values must be enclosed in quotes and lists of values must be lists of character strings in square brackets.

setup.cfg

The file format of the `setup.cfg` corresponds to an `.ini` file:

```
[tool:pytest]
addopts =
    --strict-markers
    --strict-config
    -ra
testpaths = tests
markers =
    smoke: Small subset of all tests
    exception: Only run expected exceptions
```

The only difference between this and `pytest.ini` is the specification of the `[tool:pytest]` section.

> **Warning:** However, the parser of the `.cfg` file differs from the parser of the `.ini` file, and this difference can cause problems that are difficult to track down, see also pytest documentation.

Set `rootdir`

Before pytest searches for test files to execute, it reads the configuration file `pytest.ini`, `tox.ini`, `pyproject.toml` or `setup.cfg`, which contains a pytest section:

- if you have specified a `test` directory, pytest will start searching there
- if you have specified several files or directories, pytest starts with the parent directory
- if you do not specify a file or directory, pytest starts in the current directory.

If pytest finds a configuration file in the start directory, this is the root and if not, pytest goes up the directory tree until it finds a configuration file that contains a pytest section. Once pytest has found a configuration file, it marks the directory in which it found it as `rootdir`. This root directory is also the relative root of the IDs. pytest also tells you where it has found a configuration file. Using these rules, we can run tests at different levels and be sure that pytest finds the correct configuration file:

```
$ cd items
$ pytest
============================= test session starts ==============================
...
rootdir: /Users/veit/cusy/prj/items
configfile: pyproject.toml
testpaths: tests
plugins: Faker-19.11.0
collected 39 items
...
```

`conftest.py` for sharing local fixtures and hook functions

The `conftest.py` file is used to store fixtures and hook functions, see also *Test fixtures* and *Plugins*. You can have as many `conftest.py` files in a project as you like. Everything that is defined in a `conftest.py` file applies to tests in this directory and all subdirectories. If you have a `conftest.py` file at the top test level, the fixtures defined there can be used for all tests. If there are special fixtures that only apply to a subdirectory, these can be defined in another conftest.py file in this subdirectory. For example, the CLI tests may require different fixtures than the API tests, and you can also share some of them.

Tip: However, it is a good idea to keep only one `conftest.py` file so that you can easily find the fixture definitions. Even though we can always find out where a fixture is defined with `pytest --fixtures -v`, it is still easier if it is always defined in the one `conftest.py` file.

`__init__.py` to avoid collision of test file names

The `__init__.py` file allows you to have duplicate test filenames. If you have `__init__.py` files in each test subdirectory, you can use the same test filename in multiple directories, for example:

```
items
├── ...
├── pytest.ini
├── src
│   └── ...
│
```

(continues on next page)

(continued from previous page)

```
└── tests
    ├── api
    │   ├── __init__.py
    │   └── test_add.py
    ├── cli
    │   ├── __init__.py
    │   ├── conftest.py
    │   └── test_add.py
    └── conftest.py
```

Now we can test the `add` functionality both via the API and via the CLI, whereby a `test_add.py` is located in both directories:

```
$ pytest
============================= test session starts ==============================
...
rootdir: /Users/veit/cusy/prj/items
configfile: pyproject.toml
testpaths: tests
plugins: Faker-19.11.0
collected 6 items

tests/api/test_add.py ....                                               [ 66%]
tests/cli/test_add.py ..                                                 [100%]

============================== 6 passed in 0.03s ===============================
```

Most of my projects start with the following configuration:

```
addopts =
   --strict-markers
   --strict-config
   -ra
```

See also:

- Configuration
- Configuration Options

Debugging test failures

When tests fail, we need to find out why. Maybe it's the test, or maybe it's the application. The process of finding out where the problem is and what to do about it is similar.

pytest offers many tools that can help us solve a problem faster without having to run a debugger. Python includes a built-in source code debugger called `pdb`, as well as several options that make debugging with `pdb` quick and easy.

Below we will debug some broken code using pytest options and `pdb`, looking at the debugging options and integration of pytest and `pdb`.

Debugging with pytest options

pytest contains a whole range of command line options that are useful for debugging. We will use some of them to fix our test errors. Options for selecting which tests to run, in what order, and when to stop them.

In all of these descriptions, the term *error* refers to a failed `assertion` or other uncaught `exception` found in our source or test code, including fixtures.

1. Re-execution of failed tests

   Let's start debugging by making sure that the tests fail when we re-execute them. To do this, we use `--lf` to re-execute only the failed tests and `--tb=no` to hide the traceback. This way we know that we can reproduce the error.

   1. Now we can start debugging the first error by running the first failed test, stopping after the error and looking at the traceback: `pytest --lf -x`.
   2. To make sure we understand the problem, we can run the same test again with `-l`/`--showlocals`. We don't need the full traceback again, so we can shorten it with `--tb=short`: `pytest --lf -x -l --tb=short`.

      `-l`/`--showlocals` are often very helpful and sometimes good enough to recognise a test error completely.

2. Debugging with pdb

   PDB (Python Debugger) is part of the Python standard library, so we don't need to install anything to use it. You can start pdb from pytest in several ways:

   - add a `breakpoint()` call to either the test or application code. When a pytest run encounters a `breakpoint()` function call, it will stop there and start pdb.
   - use the `--pdb` option. With `--pdb`, pytest will stop at the point of failure.
   - uses the combination of the `--lf` and `--trace` options. With `--trace` pytest stops at the beginning of each test.

     The common commands recognised by pdb are listed below:

Options	Description
Meta commands	
`h(elp)`	outputs a list of commands.
`h(elp)` *COMMAND*	outputs the help for a command.
`q(uit)`	terminates pdb.
See where you are	
`l(ist)`	lists eleven lines around the current line; when called again, the next eleven lines are listed.
`l(ist) .`	The same as above, but with a dot. Lists eleven lines around the current line. Useful if you have used l(list) a few times and have lost your current position.
`l(ist)` `first|last`	lists a specific group of lines.
`ll`	lists the entire source code for the current function.
`w(here)`	outputs the stack trace.
View values	
`p(rint)` *EXPR*	evaluates *EXPR* and outputs the value.
`pp` *EXPR*	corresponds to `p(rint)` *EXPR*, but uses `pretty-print` from the pprint module.
`a(rgs)`	outputs the argument list of the current function.
Execution commands	
`s(tep)`	executes the current line and jumps to the next line in your source code, even if it is inside a function.
`n(ext)`	executes the current line and jumps to the next line in the current function.
`c(ontinue)`	continues to the next breakpoint. When used with `--trace` , continues to the start of the next test.
`unt(il)` *LINENO*	continues to the specified line number.

See also:

The complete list can be found in Debugger Commands of the pdb documentation.

Combining pdb and tox

In order to combine pdb with tox, we need to make sure that we can pass arguments through tox to pytest. This is done with the `{posargs}` function of tox, which was described in *Passing pytest parameters to tox*. We have already set up this function in our `tox.ini` for Items:

```
[tox]
envlist = py38, py39, py310, py311
isolated_build = True
skip_missing_interpreters = True

[testenv]
deps =
  pytest
  faker
  pytest-cov
commands = pytest --cov=items --cov-fail-under=99  {posargs}
```

(continues on next page)

(continued from previous page)

```
[gh-actions]
python =
  3.8: py38
  3.9: py39
  3.10: py310
  3.11: py311
```

We want to run the Python 3.11 environment and start the debugger on a failed test with `tox -e py311 -- --pdb --no-cov`. This will take us to the pdb, right at the assertion that failed.

Once we have found and fixed the error, we can run the tox environment again with this one test error: `tox -e py311 -- --lf --tb=no --no-cov`.

Overview of the most common pytest debugger options

Options	Description
Options for selecting which tests are to be executed in which order and when they are to be stopped:	
`--lf`, `--last-failedlf`	executes the test that failed first
`--ff`, `--failed-first`	starts with the test that failed first and then executes all of them.
`-x`, `--exitfirst`	stops at the first error and then executes all.
`-maxfail=`*`NUM`*	stops the tests after *`NUM`* errors.
`--nf`, `--new-first`	executes new test files first, then the rest sorted by modification date.
`--sw`, `--stepwise`	executes the last failed test, then stops at the next error and starts again at the last failed test the next time. Similar to the combination of `--lf -x`, but more efficient.
`--sw-skip`, `--stepwise-skip`	as above, but a failed test is skipped.
Options to control pytest output:	
`-v`, `--verbose:`	verbos, `-vv` even more detailed
`--tb`	Traceback style: `[auto|long|short|line|native|no]` I usually use `--tb=short` as the default setting in the configuration file and the others for debugging.
`-l`, `--showlocals`	shows local variables next to the stacktrace.
Options to start a command line debugger:	
`--pdb`	starts the Python debugger in the event of an error. Very useful for debugging with *tox*.
`--trace`	starts the pdb source code debugger immediately when each test is executed.
`--pdbcls`	uses alternatives to pdb, for example the IPython debugger with `--pdb-cls = IPython.terminal.debugger:TerminalPdb`

16.6 Coverage

We have created an initial list of test cases. The tests in the `tests/api` directory test Items via the API. But how do we know whether these tests comprehensively test our code? This is where code coverage comes into play.

Tools that measure code coverage observe your code while a test suite is running and record which lines are passed and which are not. This measure – known as line coverage – is calculated by dividing the total number of executed lines by the total number of lines of code. Code coverage tools can also tell you whether all paths in control statements are traversed, a measurement known as branch coverage.

However, code coverage cannot tell you if your test suite is good; it can only tell you how much of the application code is being traversed by your test suite.

Coverage.py is the favourite Python tool that measures code coverage. And pytest-cov is a popular *pytest plugin* that is often used in conjunction with Coverage.py.

16.6.1 Using Coverage.py with pytest-cov

Both Coverage.py and pytest-cov are third-party packages that must be installed before use:

You can create a report for the test coverage with Coverage.py.

```
$ bin/python -m pip install coverage pytest-cov
```

```
C:> Scripts\python -m pip install coverage pytest-cov
```

Note: If you want to determine the test coverage for Python 2 and Python<3.6, you must use Coverage<6.0.

To run tests with Coverage.py, you need to add the `--cov` option and specify either a path to the code you want to measure or the installed package you are testing. In our case, the Items project is an installed package, so we will test it with `--cov=items`.

The normal pytest output is followed by the coverage report, as shown here:

```
$ cd /PATH/TO/items
$ python3 -m venv .
$ . bin/activate
$ python -m pip install ".[dev]"
$ pytest --cov=items
============================= test session starts ==============================
...
rootdir: /Users/veit/cusy/prj/items
configfile: pyproject.toml
testpaths: tests
plugins: cov-4.1.0, Faker-19.11.0
collected 35 items

tests/api/test_add.py ....                                               [ 11%]
tests/api/test_config.py .                                               [ 14%]
tests/api/test_count.py ...                                              [ 22%]
tests/api/test_delete.py ...                                             [ 31%]
tests/api/test_finish.py ....                                            [ 42%]
tests/api/test_list.py .........                                         [ 68%]
tests/api/test_start.py ....                                             [ 80%]
tests/api/test_update.py ....                                            [ 91%]
tests/api/test_version.py .                                              [ 94%]
tests/cli/test_add.py ..                                                 [100%]

---------- coverage: platform darwin, python 3.11.5-final-0 ----------
Name                    Stmts   Miss  Cover
-------------------------------------------
src/items/__init__.py       3      0   100%
```

(continues on next page)

(continued from previous page)

```
src/items/api.py            70      1    99%
src/items/cli.py            38      9    76%
src/items/db.py             23      0   100%
-----------------------------------------
TOTAL                      134     10    93%


============================== 35 passed in 0.11s ==============================
```

The previous output was generated by coverage's reporting functions, although we did not call coverage directly. `pytest --cov=items` instructed the `pytest-cov` plugin to

- set `coverage` to `items` with `--source` while running pytest with the tests
- execute `coverage` report for the line coverage report

Without `pytest-cov`, the commands would look like this:

```
$ coverage run --source=items -m pytest
$ coverage report
```

The files `__init__.py` and `db.py` have a coverage of 100%, which means that our test suite hits every line in these files. However, this does not tell us that it is sufficiently tested or that the tests detect possible errors; but it at least tells us that every line was executed during the test suite.

The `cli.py` file has a coverage of 76%. This may seem surprisingly high as we have not tested the CLI at all. However, this is due to the fact that `cli.py` is imported by `__init__.py`, so that all function definitions are executed, but none of the function contents.

However, we are really interested in the `api.py` file with 99% test coverage. We can find out what was missed by re-running the tests and adding the `--cov-report=term-missing` option:

```
pytest --cov=items --cov-report=term-missing
============================== test session starts ==============================
...
rootdir: /Users/veit/cusy/prj/items
configfile: pyproject.toml
testpaths: tests
plugins: cov-4.1.0, Faker-19.11.0
collected 35 items

tests/api/test_add.py ....                                               [ 11%]
tests/api/test_config.py .                                               [ 14%]
tests/api/test_count.py ...                                              [ 22%]
tests/api/test_delete.py ...                                             [ 31%]
tests/api/test_finish.py ....                                            [ 42%]
tests/api/test_list.py .........                                         [ 68%]
tests/api/test_start.py ....                                             [ 80%]
tests/api/test_update.py ....                                            [ 91%]
tests/api/test_version.py .                                              [ 94%]
tests/cli/test_add.py ..                                                 [100%]

---------- coverage: platform darwin, python 3.11.5-final-0 ----------
Name                    Stmts   Miss  Cover   Missing
-------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
src/items/__init__.py       3      0   100%
src/items/api.py           68      1    99%   52
src/items/cli.py           38      9    76%   18-19, 25, 39-43, 51
src/items/db.py            23      0   100%
-----------------------------------------------------
TOTAL                     132     10    92%


============================== 35 passed in 0.11s ==============================
```

Now that we have the line numbers of the untested lines, we can open the files in an editor and view the missing lines. However, it is easier to look at the HTML report.

See also:

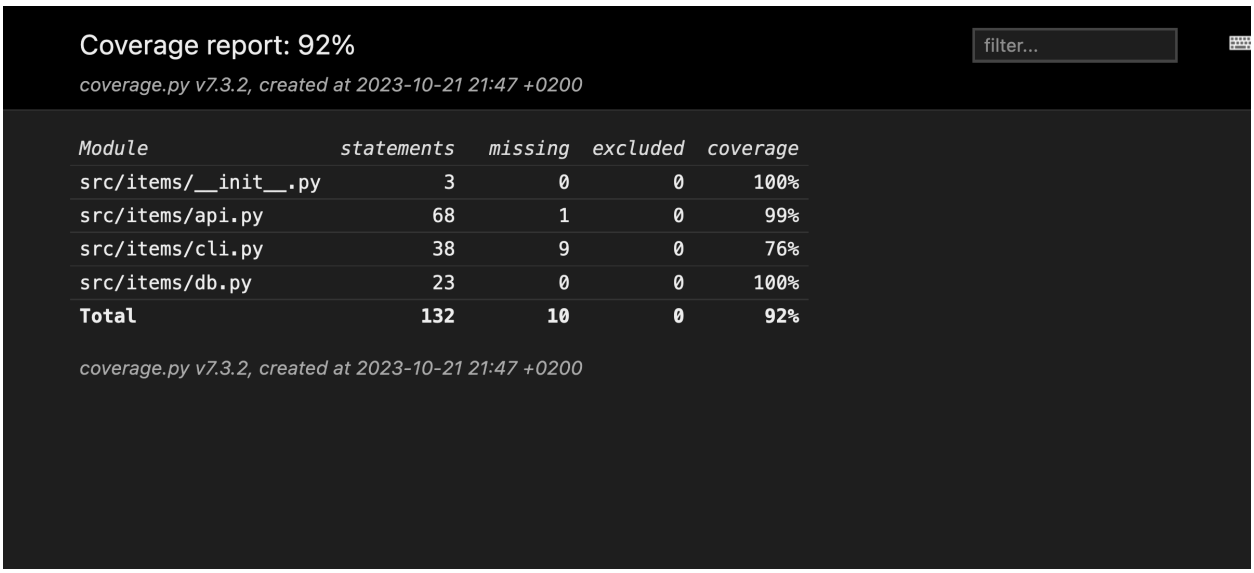
- pytest-cov's documentation

Generate HTML reports

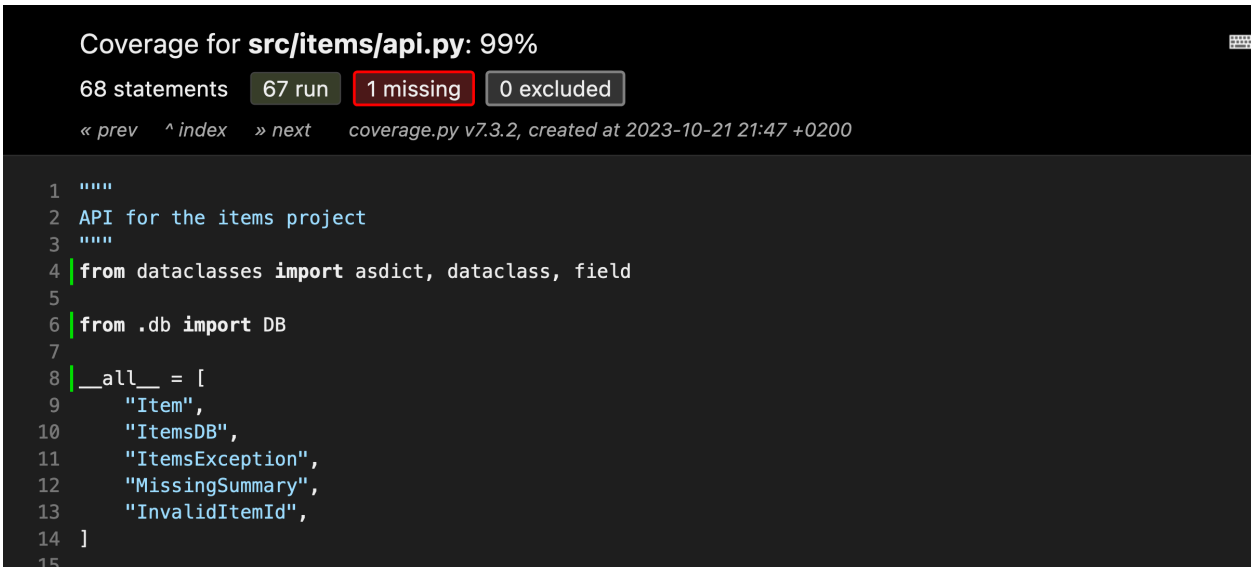
With Coverage.py we can generate HTML reports to view the coverage data in more detail. The report is generated either with the option `--cov-report=html` or by executing coverage html after a previous coverage run:

```
$ cd /PATH/TO/items
$ python3 -m venv .
$ . bin/acitvate
$ python -m pip install ".[dev]"
$ pytest --cov=items --cov-report=html
```

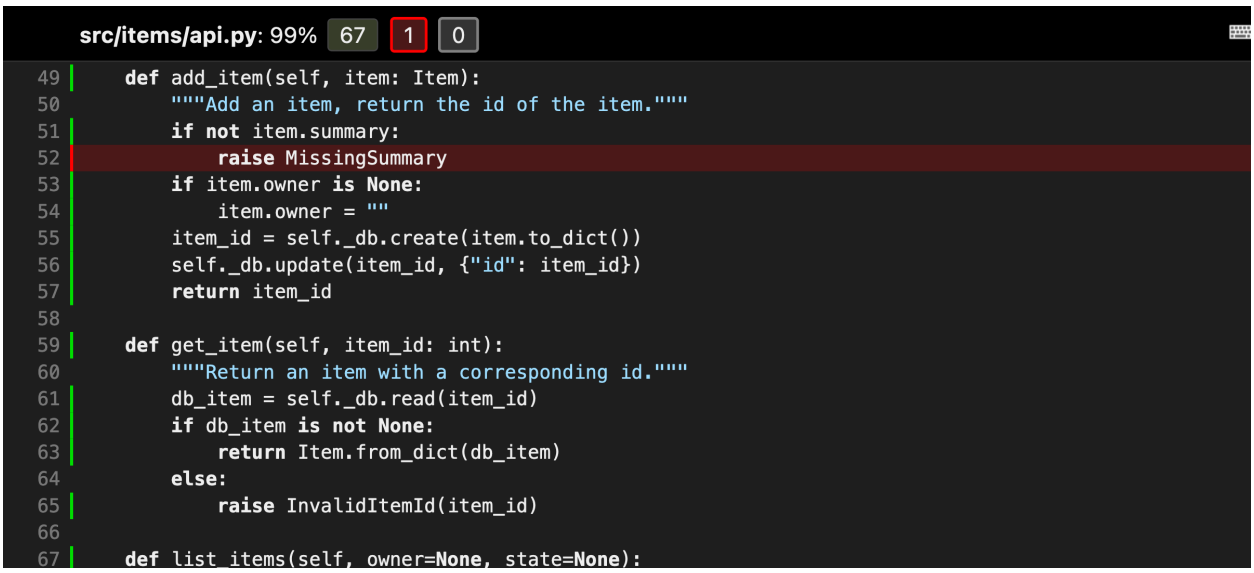
Both commands will prompt Coverage.py to create an HTML report in the `htmlcov/` directory. Open `htmlcov/index.html` with a browser and you should see the following:



If you click on the `src/items/api.py:` file, a report for this file is displayed:



The upper part of the report shows the percentage of rows covered (99%), the total number of statements (68) and how many statements were executed (67), missed (1) and excluded (0). Click on *missing* to highlight the rows that were not executed:



It looks like the function `add_item()` has an exception `MissingSummary`, which is not tested yet.

Exclude code from test coverage

In the HTML reports you will find a column with the specification *0 excluded*. This refers to a function of Coverage.py that allows us to exclude some lines from the check. We do not exclude anything in items. However, it is not uncommon for some lines of code to be excluded from the test coverage calculation, for example modules that are to be both imported and executed directly may contain a block that looks something like this:

```
if __name__ == '__main__':
    main()
```

This command tells Python to execute `main()` when we call the module directly with `python my_module.py`, but

not to execute the code when the module is imported. These types of code blocks are often excluded from testing with a simple pragma statement:

```
if __name__ == '__main__': # pragma: no cover
    main()
```

This instructs Coverage.py to exclude either a single line or a block of code. If, as in this case, the pragma is in the `if` statement, you do not have to insert it into both lines of code.

Alternatively, this can also be configured for all occurrences:

```
[run]
branch = True

[report]
; Regexes for lines to exclude from consideration
exclude_also =

    ; Don't complain if tests don't hit defensive assertion code:
    raise AssertionError
    raise NotImplementedError

    ; Don't complain if non-runnable code isn't run:
    if __name__ == .__main__.:

ignore_errors = True

[html]
directory = coverage_html_report
```

```
[tool.coverage.run]
branch = true

[tool.coverage.report]
# Regexes for lines to exclude from consideration
exclude_also = [
    # Don't complain if tests don't hit defensive assertion code:
    "raise AssertionError",
    "raise NotImplementedError",

    # Don't complain if non-runnable code isn't run:
    "if __name__ == .__main__.:",
    ]

ignore_errors = true

[tool.coverage.html]
directory = "coverage_html_report"
```

```
[coverage:run]
branch = True

[coverage:report]
; Regexes for lines to exclude from consideration
```

(continues on next page)

(continued from previous page)

```
exclude_also =

    ; Don't complain if tests don't hit defensive assertion code:
    raise AssertionError
    raise NotImplementedError

    ; Don't complain if non-runnable code isn't run:
    if __name__ == .__main__.:

ignore_errors = True

[coverage:html]
directory = coverage_html_report
```

See also:

Configuration reference

16.6.2 Extensions

In Coverage.py plugins you will also find a number of extensions for Coverage.

16.6.3 Test coverage of all tests with GitHub actions

This instructs Coverage.py to exclude either a single line or a block of code. If, as in this case, the pragma is in the if statement, you do not have to insert it into both lines of code.

Extensions In Coverage.py plugins you will also find a number of extensions for Coverage.

16.6.4 Test coverage of all tests with GitHub actions

After you have checked the test coverage, you can upload the files as GitHub actions, for example in a `ci.yaml` as artefacts, so that you can reuse them later in other jobs:

```
      - name: "Upload coverage data"
        uses: actions/upload-artifact@v3
        with:
          name: "coverage-data"
          path: .coverage.*
          if-no-files-found: ignore
```

`if-no-files-found: ignore`
: is useful if you don't want to measure the test coverage for all Python versions in order to get results faster. You should therefore only upload the data for those elements of your matrix that you want to take into account.

After all the tests have been run, you can define another job that summarises the results:

```
  coverage:
    name: "Combine and check coverage"
    needs: tests
    runs-on: ubuntu-latest
```

(continues on next page)

(continued from previous page)

```
    steps:
      - name: "Check out the repo"
        uses: "actions/checkout@v3"

      - name: "Set up Python"
        uses: "actions/setup-python@v4"
        with:
          python-version: "3.11"

      - name: "Install dependencies"
        run: |
          python -m pip install --upgrade coverage[toml]

      - name: "Download coverage data"
        uses: actions/download-artifact@v3
        with:
          name: "coverage-data"

      - name: "Combine coverage and fail it it's under 100 %"
        run: |
          python -m coverage combine
          python -m coverage html --skip-covered --skip-empty

          # Report and write to summary.
          python -m coverage report | sed 's/^/    /' >> $GITHUB_STEP_SUMMARY

          # Report again and fail if under 100%.
          python -Im coverage report --fail-under=100

      - name: "Upload HTML report if check failed"
        uses: actions/upload-artifact@v3
        with:
          name: html-report
          path: htmlcov
        if: ${{ failure() }}
```

`needs: tests`
: ensures that all tests are performed. If your job that runs the tests has a different name, you will need to change it here.

`name: "Download coverage data"`
: downloads the test coverage data that was previously uploaded with `name: "Upload coverage data"`.

`name: "Combine coverage and fail it it's under 100 %"`
: combines the test coverage and creates an HTML report if the condition `--fail-under=100` is met.

Once the workflow is complete, you can download the HTML report under *YOUR_REPO → Actions → tests → Combine and check coverage*.

See also:

- How to Ditch Codecov for Python Projects
- structlog main.yml

16.6.5 Badge

You can use GitHub Actions to create a badge with your code coverage. A GitHub Gist is also required to store the parameters for the badge, which is rendered by shields.io. To do this, we extend our `ci.yaml` as follows:

```
94      - name: "Create badge"
95        uses: schneegans/dynamic-badges-action@v1.6.0
96        with:
97          auth: ${{ secrets.GIST_TOKEN }}
98          gistID: YOUR_GIST_ID
99          filename: covbadge.json
100         label: Coverage
101         message: ${{ env.total }}%
102         minColorRange: 50
103         maxColorRange: 90
104         valColorRange: ${{ env.total }}
```

Line 97
`GIST_TOKEN` is a personal GitHub access token.

Line 98
You should replace `YOUR_GIST_ID` with your own Gist ID. If you don't have a Gist ID yet, you can create one with:

1. Call up https://gist.github.com and create a new gist, which you can name `test.json`, for example. The ID of the gist is the long alphanumeric part of the URL that you need here.
2. Then go to https://github.com/settings/tokens and create a new token with the gist area.
3. Finally, go to *YOUR_REPO* → *Settings* → *Secrets* → *Actions* and add this token. You can give it any name you like, for example *GIST_SECRET*.

   If you use Dependabot to automatically update the dependencies of your repository, you must also add the *GIST_SECRET* in *YOUR_REPO* → *Settings* → *Secrets* → *Dependabot*.

Lines 102-104
The badge is automatically coloured:

- 50 % in red
- 90 % in green
- with a colour gradient between the two

Now the badge can be displayed with a URL like this: `https://img.shields.io/endpoint?url=https://gist.githubusercontent.com/YOUR_GITHUB_NAME/GIST_SECRET/raw/covbadge.json`.

16.7 Mock

In this chapter, we will test the CLI. For this, we will use the mock package, which has been delivered as part of the Python standard library under the name `unittest.mock` since Python 3.3. For older versions of Python, you can install it with :

```
$ . bin/activate
$ python -m pip install mock
```

```
C:> Scripts\activate.bat
C:> python -m pip install mock
```

Mock objects are sometimes also referred to as test doubles, *fakes* or *stubs*. With pytest's own monkeypatch fixture and mock, you should have all the functions you need.

16.7.1 Example

Firstly, we wanted to start with a simple example and check whether the working days from Monday to Friday are determined correctly.

1. We import `datetime.datetime` and `Mock`:

```
from datetime import datetime
from unittest.mock import Mock
```

2. Then we define two test days:

```
monday = datetime(year=2021, month=10, day=11)
saturday = datetime(year=2021, month=10, day=16)
```

3. Now we define a method for checking the working days, whereby the Python datetime library treats Mondays as `0` and Sundays as `6`:

```
def is_workingday():
    today = datetime.today()
    return 0 <= today.weekday() < 5
```

4. Then we mock datetime:

```
datetime = Mock()
```

5. Finally, we test our two mock objects:

```
datetime.today.return_value = monday
# Test Tuesday is a weekday
assert is_workingday()
```

```
datetime.today.return_value = saturday
# Test Saturday is not a weekday
assert not is_workingday()
```

16.7.2 Testing with Typer

For testing the Items CLI, we will also look at how the `CliRunner` provided by Typer helps with testing. Typer provides a test interface that allows us to call our application without having to rely on `subprocess.run()` as in the short *capsys* example. This is good because we cannot simulate what is running in a separate process. So in `tests/cli/conftest.py` we can just pass our application `items.cli.app` and a list of strings representing the command to the `invoke()` function of our `runner`: more precisely, we use `shlex.split(command_string)()` to convert the commands, for example `list -o "veit"` into `["list", "-o", "veit"]` and can then intercept and return the output.

```
import shlex

import pytest
from typer.testing import CliRunner

import items

runner = CliRunner()


@pytest.fixture()
def items_cli(db_path, monkeypatch, items_db):
    monkeypatch.setenv("ITEMS_DB_DIR", db_path.as_posix())

    def run_cli(command_string):
        command_list = shlex.split(command_string)
        result = runner.invoke(items.cli.app, command_list)
        output = result.stdout.rstrip()
        return output

    return run_cli
```

We can then simply use this fixture to test the version in `tests/cli/test_version.py`, for example:

```
import items


def test_version(items_cli):
    assert items_cli("version") == items.__version__
```

16.7.3 Mocking of attributes

Let's take a look at how we can use mocking to ensure that, for example, three-digit version numbers of `items.__version__()` are also output correctly via the CLI. For this we will use `mock.patch.object()` as a context manager:

```
from unittest import mock

import items


def test_mock_version(items_cli):
    with mock.patch.object(items, "__version__", "100.0.0"):
        assert items_cli("version") == items.__version__
```

In our test code, we import `items`. The resulting items object is what we will patch. The call to `mock.patch.object()`, which is used as a *context manager* within a `with` block, returns a mock object that is cleaned up after the `with` block:

1. In this case, the `__version__` attribute of `items` is replaced with `"100.0.0"` for the duration of the `with` block.
2. We then use `items_cli()` to call our CLI application with the `"version"` command. However, when the `version()` method is called, the `__version__` attribute is not the original string, but the string we replaced with `mock.patch.object()`.

16.7.4 Mocking classes and methods

In `src/items/cli.py` we have defined `config()` as follows:

```
def config():
    """List the path to the Items db."""
    with items_db() as db:
        print(db.path())
```

`items_db()` is a *context manager* that returns an `items.ItemsDB` object. The returned object is then used as a `db` to call `db.path()`. So we should mock two things here: `items.ItemsDB` and one of its methods, `path()`. Let's start with the class:

```
from unittest import mock

import items


def test_mock_itemsdb(items_cli):
    with mock.patch.object(items, "ItemsDB") as MockItemsDB:
        mock_db_path = MockItemsDB.return_value.path.return_value = "/foo/"
        assert items_cli("config") == str(mock_db_path)
```

Let's make sure that it really works:

```
$ pytest -v -s tests/cli/test_config.py::test_mock_itemsdb
============================= test session starts ==============================
...
configfile: pyproject.toml
plugins: cov-4.1.0, Faker-19.11.0
collected 1 item

tests/cli/test_config.py::test_mock_itemsdb PASSED

============================== 1 passed in 0.04s ===============================
```

Great, now we just have to move the mock for the database to a fixture, because we will need it in many test methods:

```
@pytest.fixture()
def mock_itemsdb():
    with mock.patch.object(items"ItemsDB") as MockItemsDB:
        yield MockItemsDB.return_value
```

This fixture mocks the `ItemsDB` object and returns the `return_value` so that tests can use it to replace things like `path`:

```
def test_mock_itemsdb(items_cli, mock_itemsdb):
    mock_itemsdb.path.return_value = "/foo/"
    result = runner.invoke(app, ["config"])
    assert result.stdout.rstrip() == "/foo/"
```

Alternatively, the `@mock.patch()` decorator can also be used to mock classes or objects. In the following examples, the output of `os.listdir` is mocked. This does not require `db_path` to be present in the file system:

```
import os
from unittest import mock


@mock.patch("os.listdir", mock.MagicMock(return_value="db_path"))
def test_listdir():
    assert "db_path" == os.listdir()
```

Another alternative is to define the return value separately:

```
@mock.patch("os.listdir")
def test_listdir(mock_listdir):
    mock_listdir.return_value = "db_path"
    assert "db_path" == os.listdir()
```

16.7.5 Synchronising mocks with `autospec`

Mock objects are usually intended as objects that are used instead of the real implementation. By default, however, they will accept any access. For example, if the real object allows `start(index)()`, our mock objects should also allow `start(index)()`. However, there is a problem with this. Mock objects are too flexible by default: they would also accept `stort()` or other misspelled, renamed or deleted methods or parameters. Over time, this can lead to so-called mock drift if the interface you are modelling changes, but your mock in your test code does not. This form of mock drift can be solved by adding `autospec=True` to the mock during creation:

```
@pytest.fixture()
def mock_itemsdb():
    with mock.patch.object(items"ItemsDB", autospec=True) as MockItemsDB:
        yield MockItemsDB.return_value
```

Usually, this protection is always built in with `autospec`. The only exception I know of is if the class or object being mocked has dynamic methods or if attributes are added at runtime.

See also:

The Python documentation has a large section on `autospec`: Autospeccing.

16.7.6 Check call with `assert_called_with()`

So far, we have used the return values of a mocking method to ensure that our application code handles the return values correctly. But sometimes there is no useful return value, for example with `items add some tasks -o veit`. In these cases, we can ask the mock object if it was called correctly. After calling `items_cli("add some tasks -o veit")()`, the API is not used to check whether the item has entered the database, but a mock is used to ensure that the CLI has called the API method correctly. Finally, the implementation of the `add()` function calls `db.add_item()` with an `Item` object:

```
def test_add_with_owner(mock_itemsdb, items_cli):
    items_cli("add some task -o veit")
    expected = items.Item("some task", owner="veit", state="todo")
    mock_itemsdb.add_item.assert_called_with(expected)
```

If `add_item()` is not called or is called with the wrong type or the wrong object content, the test fails. For example, if we capitalise the string `"Veit"` in `expected`, but not in the CLI call, we get the following output:

```
$ pytest -s tests/cli/test_add.py::test_add_with_owner
============================= test session starts ==============================
...
configfile: pyproject.toml
plugins: cov-4.1.0, Faker-19.11.0
collected 1 item

tests/cli/test_add.py F
...
>           raise AssertionError(_error_message()) from cause
E           AssertionError: expected call not found.
E           Expected: add_item(Item(summary='some task', owner='Veit', state='todo',␣
↪id=None))
E           Actual: add_item(Item(summary='some task', owner='veit', state='todo',␣
↪id=None))
...
=========================== short test summary info ============================
FAILED tests/cli/test_add.py::test_add_with_owner - AssertionError: expected call not␣
↪found.
============================== 1 failed in 0.08s ===============================
```

See also:

There is a whole range of variants of `assert_called()`. A complete list and description can be found in unittest.mock.Mock.assert_called.

If the only way to test is to ensure the correct call, the various assert_called*() methods fulfil their purpose.

Wenn die einzige Möglichkeit zum Testen darin besteht, den korrekten Aufruf sicherzustellen, erfüllen die verschiedenen `assert_called*()`-Methoden ihren Zweck.

16.7.7 Create error conditions

Let's now check if the Items CLI handles error conditions correctly. For example, here is the implementation of the delete command:

```
@app.command()
def delete(item_id: int):
    """Remove item in db with given id."""
    with items_db() as db:
        try:
            db.delete_item(item_id)
        except items.InvalidItemId:
            print(f"Error: Invalid item id {item_id}")
```

To test how the CLI handles an error condition, we can pretend that `delete_item()` generates an exception by assigning the exception to the side_effect attribute of the mock object, like this:

```
def test_delete_invalid(mock_itemsdb, items_cli):
    mock_itemsdb.delete_item.side_effect = items.api.InvalidItemId
    out = items_cli("delete 42")
    assert "Error: Invalid item id 42" in out
```

That's all we need to test the CLI: mocking return values, checking calls to mock functions and mocking exceptions. However, there is a whole range of other mocking techniques that we have not covered. So be sure to read unittest.mock — mock object library if you want to use mocking extensively.

16.7.8 Limitations of mocking

One of the biggest problems with using mocks is that we are no longer testing the behaviour in a test, but the implementation. However, this is not only time-consuming but also dangerous: a valid refactoring, for example changing a variable name, can cause tests to fail if that particular variable has been mocked. However, we only want our tests to fail when there are breaks in behaviour, not just when there are code changes.

However, sometimes mocking is the easiest way to create exceptions or error conditions and make sure your code handles them correctly. There are also cases where testing behaviour is unreasonable, such as when accessing a payment API or sending emails. In these cases, a good option is to test whether your code calls a specific API method at the right time and with the right parameters.

See also:

- Hynek Schlawack: "Don't Mock What You Don't Own"

16.7.9 Avoid mocking with tests on multiple levels

We can also test the Items CLI without mocks by also using the API. We will not test the API, but only use it to check the behaviour of actions that are executed via the CLI. We can also test the *test_add_with_owner* example as follows:

```
def test_add_with_owner(items_db, items_cli):
    items_cli("add some task -o veit")
    expected = items.Item("some task", owner="veit", state="todo")
    all = items_db.list_items()
    assert len(all) == 1
    assert all[0] == expected
```

Mocking tests the implementation of the command line interface and ensures that an API call is made with certain parameters. The mixed-layer approach tests the behaviour to ensure that the result meets our expectations. This approach is much less of a change detector and has a greater chance of remaining valid during a refactoring. Interestingly, the tests are also about twice as fast:

```
$ pytest -s tests/cli/test_add.py::test_add_with_owner
============================= test session starts ==============================
...
configfile: pyproject.toml
plugins: cov-4.1.0, Faker-19.11.0
collected 1 item

tests/cli/test_add.py .

============================== 1 passed in 0.03s ===============================
```

We could also avoid mocking in another way. We could test the behaviour completely via the CLI. This might require parsing the output of the items list to check the correct database content.

In the API, `add_item()` returns an index and provides a `get_item(index)()` method to help with testing. Both methods are not available in the CLI, but could be. We could perhaps add the `items get index` or `items info index` commands so we can retrieve an item instead of having to use `items list` for everything. `list` also already supports filtering. Maybe filtering by index would work instead of adding a new command. And we could add an output to `items add` that says something like *Item added at index 3*. These changes would fall into the *Design for Testability* category. They also don't seem to be deep interface interventions and perhaps should be considered in future versions.

16.7.10 Plugins to support mocking

So far we have focussed on the direct use of mock. However, there are many plugins that help with mocking, such as pytest-mock, which provides a `mocker` fixture. One advantage is that the fixture cleans up after itself, so you don't need to use a `with` block like we did in our examples.

There are also some special mocking libraries:

- The following are suitable for mocking database accesses:
  - pytest-postgresql
  - pytest-mongo
  - pytest-mysql
  - pytest-dynamodb.
- You can use pytest-httpserver to test HTTP servers.
- You can use responses or betamax to mock requests.
- Other tools for different requirements are:
  - pytest-rabbitmq
  - pytest-solr
  - pytest-elasticsearch und pytest-redis.

16.8 tox

tox is an automation tool that works similarly to a *CI* tool, but can be run both locally and in conjunction with other CI tools on a server.

In the following, we will set up tox for our Items application so that it helps us with local testing. We will then set up testing using GitHub Actions.

16.8.1 Introduction to tox

tox is a command line tool that allows you to run your complete test suite in different environments. We will use tox to test the Items project in multiple Python versions, but tox is not limited to Python versions only. You can use it to test with different dependency configurations and different configurations for different operating systems. tox uses project information from the `setup.py` or `pyproject.toml` file for the package under test to create an installable *distribution of your package*. It searches the `tox.ini` file for a list of environments and then performs the following steps for each:

1. creates a *virtual environment*,
2. installs some dependencies with *pip*,
3. build your package,
4. install your package with pip,
5. run further tests.

After all environments have been tested, tox outputs a summary of the results.

Note: Although tox is used by many projects, there are alternatives that fulfil similar functions. Two alternatives to tox are nox and invoke.

16.8.2 Setting up tox

Until now, we had the items code in a `src/` directory and the tests in `tests/api/` and `tests/cli/`. Now we will add a `tox.ini` file so that the structure looks like this:

```
items
├── ...
├── pyproject.toml
├── src
│   └── items
│       └── ...
├── tests
│   ├── api
│   │   ├── __init__.py
│   │   ├── conftest.py
│   │   └── test_....py
│   └── cli
│       ├── __init__.py
│       ├── conftest.py
│       └── test_....py
└── tox.ini
```

This is a typical layout for many projects. Let's take a look at a simple `tox.ini` file in the Items project:

```
[tox]
envlist = py311
isolated_build = True

[testenv]
deps =
  pytest>=6.0
  faker
commands = pytest
```

In the `[tox]` section, we have defined `envlist = py311`. This is a shortcut that tells tox to run our tests with Python version 3.11. We will be adding more Python versions shortly, but using one version helps to understand the flow of tox.

Also note the line `isolated_build = True`: This is required for all packages configured with `pyproject.toml`. However, for all projects configured with `setup.py` that use the *setuptools* library, this line can be omitted.

In the `[testenv]` section, `pytest` and `faker` are listed as dependencies under `deps`. So tox knows that we need these two tools for testing. If you wish, you can also specify which version should be used, for example `pytest>=6.0`. Finally, commands instruct tox to execute `pytest` in every environment.

16.8.3 Executing tox

Before you can run tox, you must ensure that you have installed it:

```
$ python3 -m venv .
$ . bin/acitvate
$ python -m pip install tox
```

```
C:> python -m venv .
C:> Scripts\activate
C:> python -m pip install tox
```

To run tox, simply start tox:

```
$ tox
.pkg: _optional_hooks> python /PATH/TO/items/lib/python3.11/site-packages/pyproject_api/_
↪backend.py True hatchling.build
.pkg: get_requires_for_build_sdist> python PATH/TO/items/lib/python3.11/site-packages/
↪pyproject_api/_backend.py True hatchling.build
.pkg: build_sdist> python PATH/TO/items/lib/python3.11/site-packages/pyproject_api/_
↪backend.py True hatchling.build
py311: install_package> python -I -m pip install --force-reinstall --no-deps PATH/TO/
↪items/.tox/.tmp/package/14/items-0.1.0.tar.gz
py311: commands[0]> pytest
============================= test session starts ==============================
...
configfile: pyproject.toml
testpaths: tests
plugins: Faker-19.11.0
collected 49 items

tests/api/test_add.py ....                                               [  8%]
```

(continues on next page)

(continued from previous page)

```
tests/api/test_config.py .                                               [ 10%]
tests/api/test_count.py ...                                              [ 16%]
tests/api/test_delete.py ...                                             [ 22%]
tests/api/test_finish.py ....                                            [ 30%]
tests/api/test_list.py .........                                         [ 48%]
tests/api/test_start.py ....                                             [ 57%]
tests/api/test_update.py ....                                            [ 65%]
tests/api/test_version.py .                                              [ 67%]
tests/cli/test_add.py ..                                                 [ 71%]
tests/cli/test_config.py ..                                              [ 75%]
tests/cli/test_count.py .                                                [ 77%]
tests/cli/test_delete.py .                                               [ 79%]
tests/cli/test_errors.py ....                                            [ 87%]
tests/cli/test_finish.py .                                               [ 89%]
tests/cli/test_list.py ..                                                [ 93%]
tests/cli/test_start.py .                                                [ 95%]
tests/cli/test_update.py .                                               [ 97%]
tests/cli/test_version.py .                                              [100%]

============================== 49 passed in 0.08s ==============================
.pkg: _exit> python /PATCH/TO/items/lib/python3.11/site-packages/pyproject_api/_backend.
↪py True hatchling.build
  py311: OK (1.48=setup[1.21]+cmd[0.27] seconds)
  congratulations :) (1.51 seconds)
```

16.8.4 Testing multiple Python versions

To do this, we extend `envlist` in the `tox.ini` file to add further Python versions:

```
[tox]
envlist = py38, py39, py310, py311
isolated_build = True
skip_missing_interpreters = True
```

We will now test Python versions from 3.8 to 3.11. In addition, we have also added the setting `skip_missing_interpreters = True` so that tox does not fail if one of the listed Python versions is missing on your system. If the value is set to `True`, tox will run the tests with every available Python version, but will skip versions it doesn't find without failing. The output is very similar, although I will only highlight the differences in the following illustration:

```
 $ tox
 py38: skipped because could not find python interpreter with spec(s): py38
 py38: SKIP  in 2.13 seconds
 py39: install_package> python -I -m pip install --force-reinstall --no-deps /PATCH/TO/
↪items/.tox/.tmp/package/15/items-0.1.0.tar.gz
 py39: commands[0]> pytest
 ============================== test session starts ==============================
 ...
 ============================== 49 passed in 0.16s ==============================
 py39: OK ✓ in 8.08 seconds
 py310: skipped because could not find python interpreter with spec(s): py310
```

(continues on next page)

(continued from previous page)

```
 py310: SKIP  in 0 seconds
 py311: install_package> python -I -m pip install --force-reinstall --no-deps /PATH/TO/
↪items/.tox/.tmp/package/16/items-0.1.0.tar.gz
 py311: commands[0]> pytest
 ============================= test session starts ==============================
 ...
 ============================== 49 passed in 0.09s ==============================
 .pkg: _exit> python /PYTH/TO/items/lib/python3.11/site-packages/pyproject_api/_backend.
↪py True hatchling.build
   py38: SKIP (2.13 seconds)
   py39: OK (8.08=setup[6.92]+cmd[1.16] seconds)
   py310: SKIP (0.00 seconds)
   py311: OK (1.24=setup[0.95]+cmd[0.29] seconds)
   congratulations :) (11.48 seconds)
```

16.8.5 Running Tox environments in parallel

In the previous example, the different environments were executed one after the other. It is also possible to run them in parallel with the `-p` option:

```
$ tox -p
py38: SKIP  in 0.02 seconds
py310: SKIP  in 0.29 seconds
py311: OK ✓ in 1.53 seconds
  py38: SKIP (0.02 seconds)
  py39: OK (2.21=setup[1.88]+cmd[0.33] seconds)
  py310: SKIP (0.29 seconds)
  py311: OK (1.53=setup[1.24]+cmd[0.29] seconds)
  congratulations :) (2.24 seconds)
```

Note: The output is not abbreviated; this is the full output you will see if everything works.

16.8.6 Add coverage report in tox

The configuration of coverage reports can easily be added to the `tox.ini` file. To do this, we need to add `pytest-cov` to the `deps` settings so that the `pytest-cov` plugin is installed in the tox test environments. Including `pytest-cov` also includes all its dependencies, such as `coverage`. We then extend commands to `pytest --cov=items`:

```
[tox]
envlist = py38, py39, py310, py311
isolated_build = True
skip_missing_interpreters = True

[testenv]
deps =
  pytest
  faker
  pytest-cov
commands = pytest --cov=items
```

When using Coverage with tox, it can sometimes be useful to set up a `.coveragerc` file to tell Coverage which source code paths should be considered identical:

```
[paths]
source =
   src
   .tox/*/site-packages
```

The items source code is initially located in `src/items/` before tox creates the virtual environments and installs items in the environment. It is then located in `.tox/py311/lib/python3.11/site-packages/items`, for example.

```
$ tox -e py311
...
py311: commands[0]> pytest --cov=items
...
---------- coverage: platform darwin, python 3.11.5-final-0 ----------
Name                                                                 Stmts   Miss  Cover
----------------------------------------------------------------------------------------
.tox/py311/lib/python3.11/site-packages/items/__init__.py            3      0   100%
.tox/py311/lib/python3.11/site-packages/items/api.py                 68      1    99%
.tox/py311/lib/python3.11/site-packages/items/cli.py                 86      0   100%
.tox/py311/lib/python3.11/site-packages/items/db.py                  23      0   100%
----------------------------------------------------------------------------------------
TOTAL                                                               180      1    99%


============================== 49 passed in 0.17s ==============================
...
  py311: OK (1.85=setup[1.34]+cmd[0.51] seconds)
  congratulations :) (1.89 seconds)
```

Note: We have used the `-e py311` option here to select a specific environment.

16.8.7 Set minimum coverage

When executing coverage by tox, it also makes sense to define a minimum coverage level in order to recognise any coverage failures. This is achieved with the `--cov-fail-under` option:

```
[tox]
envlist = py38, py39, py310, py311
isolated_build = True
skip_missing_interpreters = True

[testenv]
deps =
  pytest
  faker
  pytest-cov
commands = pytest --cov=items --cov-fail-under=100
```

This adds an additional line to the output:

```
$ tox -e py311
...
============================= test session starts ==============================
...
---------- coverage: platform darwin, python 3.11.5-final-0 ----------
Name                                                                    Stmts   Miss  Cover
-------------------------------------------------------------------------------------------
.tox/py311/lib/python3.11/site-packages/items/__init__.py                   3      0   100%
.tox/py311/lib/python3.11/site-packages/items/api.py                       68      1    99%
.tox/py311/lib/python3.11/site-packages/items/cli.py                       86      0   100%
.tox/py311/lib/python3.11/site-packages/items/db.py                        23      0   100%
-------------------------------------------------------------------------------------------
TOTAL                                                                     180      1    99%

FAIL Required test coverage of 100% not reached. Total coverage: 99.44%

============================== 49 passed in 0.16s ==============================
py311: exit 1 (0.43 seconds) /PATH/TO/items> pytest --cov=items --cov-fail-under=100␣
↪pid=58109
.pkg: _exit> python /PATH/TO/items/lib/python3.11/site-packages/pyproject_api/_backend.
↪py True hatchling.build
  py311: FAIL code 1 (1.65=setup[1.22]+cmd[0.43] seconds)
  evaluation failed :( (1.68 seconds)
```

16.8.8 Passing pytest parameters to tox

We can also call individual tests with tox by making another change so that parameters can be passed to pytest:

```
[tox]
envlist = py38, py39, py310, py311
isolated_build = True
skip_missing_interpreters = True

[testenv]
deps =
  pytest
  faker
  pytest-cov
commands = pytest --cov=items --cov-fail-under=100  {posargs}
```

To pass arguments to pytest, insert them between the tox arguments and the pytest arguments. In this case, we select `test_version` tests with the `-k` keyword option. We also use `--no-cov` to disable coverage:

```
$ tox -e py311 -- -k test_version --no-cov
...
py311: commands[0]> pytest --cov=items --cov-fail-under=100 -k test_version --no-cov
============================= test session starts ==============================
...
configfile: pyproject.toml
testpaths: tests
plugins: cov-4.1.0, Faker-19.11.0
collected 49 items / 47 deselected / 2 selected
```

(continues on next page)

(continued from previous page)

```
tests/api/test_version.py .                                              [ 50%]
tests/cli/test_version.py .                                              [100%]

======================== 2 passed, 47 deselected in 0.04s ========================
.pkg: _exit> python /PATH/TO/items/lib/python3.11/site-packages/pyproject_api/_backend.
↪py True hatchling.build
  py311: OK (1.51=setup[1.25]+cmd[0.26] seconds)
  congratulations :) (1.53 seconds)
```

tox is not only ideal for the local automation of test processes, but also helps with server-based *CI*. Let's continue with the execution of pytest and tox using GitHub actions.

16.8.9 Running tox with GitHub actions

If your project is hosted on GitHub, you can use GitHub actions to automatically run your tests in different environments. A whole range of environments are available for GitHub actions: github.com/actions/virtual-environments.

1. To create a GitHub action in your project, click on *Actions* → *set up a workflow yourself*. This usually creates a `.github/workflows/main.yml` file.
2. Give this file a more descriptive name. We usually use `ci.yml` for this.
3. The prefilled YAML file is not very helpful for our purposes. You can replace the text, for example with:

```yaml
name: CI
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python: ["3.8", "3.9", "3.10", "3.11"]
    steps:
      - uses: actions/checkout@v2
      - name: Setup Python
        uses: actions/setup-python@v2
        with:
          python-version: ${{ matrix.python }}
      - name: Install tox and any other packages
        run: python -m pip install tox tox-gh-actions
      - name: Run tox for "${{ matrix.python }}"
        run: python -m tox
```

name
: can be any name. It is displayed in the GitHub Actions user interface.

on: [push, pull_request]
: instructs Actions to run our tests every time we either push code to the repository or a pull request is created. In the case of pull requests, the result of the test run can be viewed in the pull request interface. All results of the GitHub actions can be seen on the GitHub user interface.

runs-on: ubuntu-latest
: specifies the operating system on which the tests are to be executed. Here the tests only run on Linux, but other operating systems are also available.

`matrix: python: ["3.8", "3.9", "3.10", "3.11"]`
: specifies which Python version is to be executed.

`steps`
: is a list of steps. The name of each step can be arbitrary and is optional.

`uses: actions/checkout@v2`
: is a GitHub actions tool that checks out our repository so that the rest of the workflow can access it.

`uses: actions/setup-python@v2`
: is a GitHub actions tool that configures Python and installs it in a build environment.

`with: python-version: ${{ matrix.python }}`
: says that an environment should be created for each of the Python versions listed in `matrix.python`.

`run: python -m pip install tox tox-gh-actions`
: installs tox and simplifies the execution of tox in GitHub actions with tox-gh-actions by providing the environment that tox itself uses as the environment for the tests. However, we still need to adjust our `tox.ini` file for this, for example:

```
[gh-actions]
python =
    3.8: py38
    3.9: py39
    3.10: py310
    3.11: py311
```

This assigns GitHub actions to tox environments.

Note:

- You do not need to specify all variants of your environment. This distinguishes `tox-gh-actions` from `tox -e py`.
- Make sure that the versions in the `[gh-actions]` section match the available Python versions and, if applicable, those in the GitHub actions for Git pre-commit hooks.
- Since all tests for a specific Python version are executed one after the other in a container, the advantages of parallel execution are lost.

`run: python -m tox`
: executes tox.

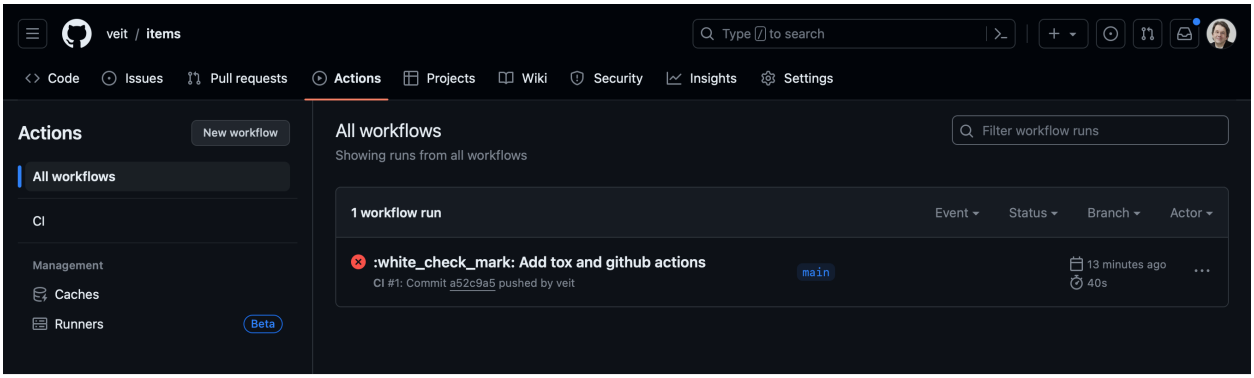
4. You can then click on *Start commit*. As we want to make further changes before the tests are executed automatically, we select *Create a new branch for this commit and start a pull request* and github-actions as the name for the new branch. Finally, you can click on *Create pull request*.
5. To switch to the new branch, we go to *Code* → *main* → *github-actions*.

The actions syntax is well documented. A good starting point in the GitHub Actions documentation is the Building and Testing Python page. The documentation also shows you how to run pytest directly without tox and how to extend the matrix to multiple operating systems. As soon as you have set up your `*.yml` file and uploaded it to your GitHub repository, it will be executed automatically. You can then see the runs in the *Actions* tab:

The different Python environments are listed on the left-hand side. If you select one, the results for this environment are displayed, as shown in the following screenshot:

See also:

- Building and testing Python



- Workflow syntax for GitHub Actions

16.8.10 Display badge

Now you can add a badge of your *CI* status to your `README.rst` file, for example with:

```
.. image:: https://github.com/YOU/YOUR_PROJECT/workflows/CI/badge.svg?branch=main
   :target: https://github.com/YOU/YOUR_PROJECT/actions?workflow=CI
   :alt: CI Status
```

16.8.11 Publish test coverage

You can publish the test coverage on GitHub, see also *Coverage GitHub-Actions*.

16.8.12 Extend tox

tox uses pluggy to customise the default behaviour. Pluggy finds a plugin by searching for an entry point with the name `tox`, for example in a `pyproject.toml` file:

```
[project.entry-points.tox]
my_plugin = "my_plugin.hooks"
```

To use the plugin, it therefore only needs to be installed in the same environment in which tox is running and it is found via the defined entry point.

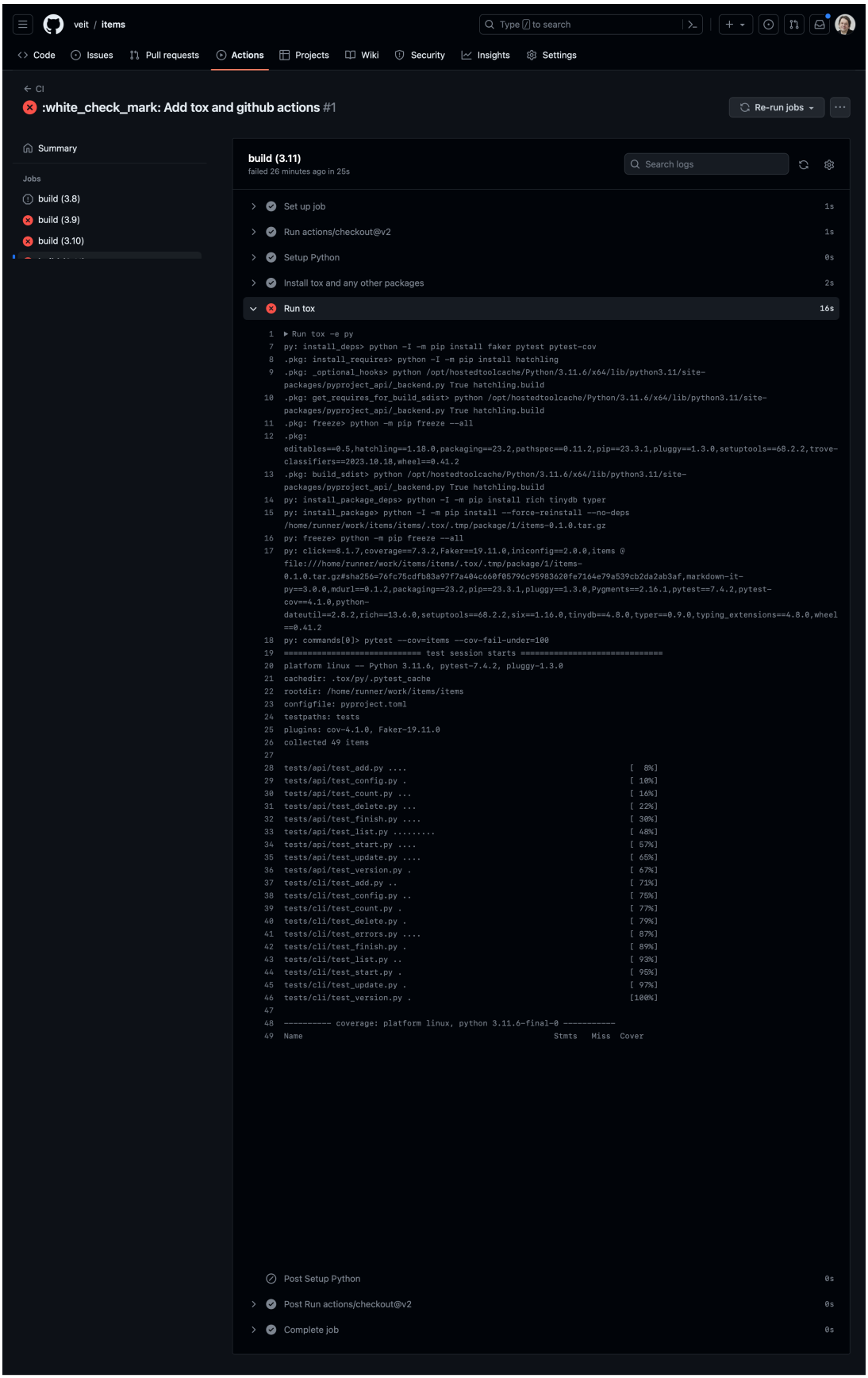
A plugin is created by implementing extension points in the form of hooks. For example, the following code snippet would define a new `--my` CLI:

```
from tox.config.cli.parser import ToxParser
from tox.plugin import impl


@impl
def tox_add_option(parser: ToxParser) -> None:
    parser.add_argument("--my", action="store_true", help="my option")
```

See also:

- Extending tox


veit / items
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
CI
:white_check_mark: Add tox and github actions #1
Re-run jobs
Summary
Jobs
build (3.8)
build (3.9)
build (3.10)
build (3.11)
failed 26 minutes ago in 25s
Search logs
Set up job 1s
Run actions/checkout@v2 1s
Setup Python 0s
Install tox and any other packages 2s
Run tox 16s
▶ Run tox -e py
py: install_deps> python -I -m pip install faker pytest pytest-cov
.pkg: install_requires> python -I -m pip install hatchling
.pkg: _optional_hooks> python /opt/hostedtoolcache/Python/3.11.6/x64/lib/python3.11/site-packages/pyproject_api/_backend.py True hatchling.build
.pkg: get_requires_for_build_sdist> python /opt/hostedtoolcache/Python/3.11.6/x64/lib/python3.11/site-packages/pyproject_api/_backend.py True hatchling.build
.pkg: freeze> python -m pip freeze --all
.pkg: editables==0.5,hatchling==1.18.0,packaging==23.2,pathspec==0.11.2,pip==23.3.1,pluggy==1.3.0,setuptools==68.2.2,trove-classifiers==2023.10.18,wheel==0.41.2
.pkg: build_sdist> python /opt/hostedtoolcache/Python/3.11.6/x64/lib/python3.11/site-packages/pyproject_api/_backend.py True hatchling.build
py: install_package_deps> python -I -m pip install rich tinydb typer
py: install_package> python -I -m pip install --force-reinstall --no-deps /home/runner/work/items/items/.tox/.tmp/package/1/items-0.1.0.tar.gz
py: freeze> python -m pip freeze --all
py: click==8.1.7,coverage==7.3.2,Faker==19.11.0,iniconfig==2.0.0,items @ file:///home/runner/work/items/items/.tox/.tmp/package/1/items-0.1.0.tar.gz#sha256=76fc75cdfb83a97f7a404c660f05796c95983620fe7164e79a539cb2da2ab3af,markdown-it-py==3.0.0,mdurl==0.1.2,packaging==23.2,pip==23.3.1,pluggy==1.3.0,Pygments==2.16.1,pytest==7.4.2,pytest-cov==4.1.0,python-dateutil==2.8.2,rich==13.6.0,setuptools==68.2.2,six==1.16.0,tinydb==4.8.0,typer==0.9.0,typing_extensions==4.8.0,wheel==0.41.2
py: commands[0]> pytest --cov=items --cov-fail-under=100
============================= test session starts ==============================
platform linux -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0
cachedir: .tox/py/.pytest_cache
rootdir: /home/runner/work/items/items
configfile: pyproject.toml
testpaths: tests
plugins: cov-4.1.0, Faker-19.11.0
collected 49 items
tests/api/test_add.py .... [8%]
tests/api/test_config.py . [10%]
tests/api/test_count.py ... [16%]
tests/api/test_delete.py ... [22%]
tests/api/test_finish.py .... [30%]
tests/api/test_list.py .... [48%]
tests/api/test_start.py .... [57%]
tests/api/test_update.py .... [65%]
tests/api/test_version.py . [67%]
tests/cli/test_add.py .. [71%]
tests/cli/test_config.py .. [75%]
tests/cli/test_count.py . [77%]
tests/cli/test_delete.py . [79%]
tests/cli/test_errors.py .... [87%]
tests/cli/test_finish.py . [89%]
tests/cli/test_list.py .. [93%]
tests/cli/test_start.py . [95%]
tests/cli/test_update.py . [97%]
tests/cli/test_version.py . [100%]
---------- coverage: platform linux, python 3.11.6-final-0 -----------
Name Stmts Miss Cover
Post Setup Python 0s
Post Run actions/checkout@v2 0s
Complete job 0s

- tox development team

16.9 unittest2

unittest2 is a backport of `unittest`, with improved API and assertions than in previous Python versions.

16.9.1 Example

You may want to import the module under the name `unittest` to make it easier to port code to newer versions of the module in the future:

```
import unittest2 as unittest

class MyTest(unittest.TestCase):
    ...
```

This way, if you switch to a newer Python version and no longer need the `unittest2` module, you can simply change the import in your test module without having to change any further code.

16.9.2 Installation

```
$ bin/python -m pip install unittest2
```

```
C:> Scripts\python -m pip install unittest2
```

16.10 Glossary

`assert`
: A keyword that stops code execution if its argument is false.

Continuous integration
CI
: Automatic checking of the creation and test process on different platforms.

Dummy
: Object that is passed around but never actually used. Normally they are only used to fill parameter lists.

`exception`
: Customisable form of *assert*.

`except`
: Keyword used to catch an *exception* and handle it carefully.

Fake
: Object that has an actual working implementation, but usually takes a shortcut that makes it unsuitable for production.

Integration test
: Tests that verify that the different parts of the software work together as expected.

Mock
: Objects that are programmed with *exception* that form a specification of the calls you are likely to receive.

  See also:

  - Mock object

pytest
: A Python package with test utilities.

Regression test
: Test to protect against new errors or regressions that can occur due to new software and updates.

Stubs
: provide ready-made responses to calls made during the test and usually do not respond at all to anything that has not been programmed for the test.

Test-driven development
TDD
: A software development strategy in which the tests are written before the code.

`try`
: A keyword that protects a part of the code that can trigger an *exception*.

CHAPTER

SEVENTEEN

DOCUMENT

In order for your software package to be useful, documentation is required that describes how your software can be installed, operated, used and improved:

- Those who want to use your package need information,
  - what problems your software solves and what the main features and limitations of the software are (`README`)
  - how the software can be used as an example
  - what changes have come in more recent software versions (`CHANGELOG`)
- Those who want to run the software need an installation guide for your software and the required dependencies.
- Those who want to improve the software need information about
  - how to help improve the product with bug fixes (`CONTRIBUTING`)
  - how to communicate with others (`CODE_OF_CONDUCT`)

All together need information on how the product is licensed (`LICENSE` file or `LICENSES` folder) and how to get help if needed.

See also:

- Eric Holscher: Why You Shouldn't Use "Markdown" for Documentation
- Tom Johnson: 10 reasons for moving away from DITA
- Tom Johnson: Jekyll versus DITA
- Google developer documentation style guide
- Google Technical Writing Courses for Engineers

17.1 Create a Sphinx project

17.1.1 Installation and start

1. Create a virtual environment for your documentation project:

```
$ python3 -m venv venv
```

```
C:> python -m venv venv
```

2. Switch to the virtual environment and install Sphinx there:

```
$ cd !$
cd venv
$ bin/python -m pip install sphinx
Creating a virtualenv for this project...
...
```

```
C:> cd venv
C:> bin/python -m pip install sphinx
Creating a virtualenv for this project...
...
```

3. Create your Sphinx documentation project:

```
$ bin/sphinx-quickstart docs
Selected root path: docs
> Separate source and build directories (y/n) [n]:
> Name prefix for templates and static dir [_]:
> Project name: my.package
> Author name(s): Veit Schiele
> Project release []: 1.0
> Project language [en]:
> Source file suffix [.rst]:
> Name of your master document (without suffix) [index]:
> autodoc: automatically insert docstrings from modules (y/n) [n]: y
> doctest: automatically test code snippets in doctest blocks (y/n) [n]: y
> intersphinx: link between Sphinx documentation of different projects (y/n) [n]: y
> todo: write "todo" entries that can be shown or hidden on build (y/n) [n]: y
> coverage: checks for documentation coverage (y/n) [n]:
> imgmath: include math, rendered as PNG or SVG images (y/n) [n]:
> mathjax: include math, rendered in the browser by MathJax (y/n) [n]:
> ifconfig: conditional inclusion of content based on config values (y/n) [n]:
> viewcode: include links to the source code of documented Python objects (y/n)␣
↪[n]: y
> githubpages: create .nojekyll file to publish the document on GitHub pages (y/n)␣
↪[n]:
> Create Makefile? (y/n) [y]:
> Create Windows command file? (y/n) [y]:

Creating file docs/source/conf.py.
Creating file docs/source/index.rst.
Creating file docs/Makefile.
Creating file docs/make.bat.
```

```
C:> Scripts\sphinx-quickstart docs
Selected root path: docs
> Separate source and build directories (y/n) [n]:
> Name prefix for templates and static dir [_]:
> Project name: my.package
> Author name(s): Veit Schiele
> Project release []: 1.0
> Project language [en]:
> Source file suffix [.rst]:
```

(continues on next page)

(continued from previous page)

```
> Name of your master document (without suffix) [index]:
> autodoc: automatically insert docstrings from modules (y/n) [n]: y
> doctest: automatically test code snippets in doctest blocks (y/n) [n]: y
> intersphinx: link between Sphinx documentation of different projects (y/n) [n]: y
> todo: write "todo" entries that can be shown or hidden on build (y/n) [n]: y
> coverage: checks for documentation coverage (y/n) [n]:
> imgmath: include math, rendered as PNG or SVG images (y/n) [n]:
> mathjax: include math, rendered in the browser by MathJax (y/n) [n]:
> ifconfig: conditional inclusion of content based on config values (y/n) [n]:
> viewcode: include links to the source code of documented Python objects (y/n)␣
↪[n]: y
> githubpages: create .nojekyll file to publish the document on GitHub pages (y/n)␣
↪[n]:
> Create Makefile? (y/n) [y]:
> Create Windows command file? (y/n) [y]:

Creating file docs\conf.py.
Creating file docs\index.rst.
Creating file docs\Makefile.
Creating file docs\make.bat.
```

17.1.2 Sphinx layout

```
venv
└── docs
    ├── Makefile
    ├── _static
    ├── _templates
    ├── conf.py
    ├── index.rst
    └── make.bat
```

`index.rst` is the initial file for the documentation, in which the table of contents is located. The table of contents will be expanded by you as soon as you add new `*.rst` files.

17.1.3 Generate the documentation

You can now generate the documentation, for example with:

```
$ bin/sphinx-build -ab html docs/ docs/_build
```

```
C:> Scripts\sphinx-build -ab html docs\ docs\_build
```

a

regenerates all pages of the documentation.

> **Note:** This is always useful if you have added new pages to your documentation. to your documentation.

b
: specifies which builder should be used to generate the documentation. In our example this is `html`.

17.2 reStructuredText

See also:

- reStructuredText Primer
- reStructuredText Quick Reference

17.2.1 Headlines

```
Underline the title with punctuation marks
==========================================

Change the punctuation mark for subtitles
-----------------------------------------
```

17.2.2 Paragraphs

A paragraph consists of one or more lines of non-indented text, separated from the material above and below by blank lines.

```
A paragraph consists of one or more lines of non-indented text, separated
from the material above and below by blank lines.
```

17.2.3 Inline markup

Italic, **bold** and `preformatted`

```
*Italic*, **bold** and ``preformatted``
```

17.2.4 Links

External links

hyperlink link

```
`hyperlink <http://en.wikipedia.org/wiki/Hyperlink>`_ `link`_

.. _link: http://en.wikipedia.org/wiki/Link_(The_Legend_of_Zelda)
```

Note: A directive starting with `..` must always be preceded by a blank line.

Internal links

Cross-referencing locations

Section to reference

To refer to the section, use *Section to reference*.

```
.. _my-reference-label:

Section to reference
....................

To refer to the section, use :ref:`my-reference-label`.
```

Referencing documents

Link to *start page* or *Docstrings*.

```
Link to :doc:`start page <../index>` or :doc:`docstrings`.
```

Download documents

Link to a document, not rendered by Sphinx, for example `docstrings-example.rst`.

```
Link to a document, not rendered by Sphinx, for example
:download:`docstrings-example.rst`.
```

17.2.5 Images

```
.. image:: uml/activity-diagram.svg
```

Other semantic markup

File listing

`/Users/NAME/python-basics`

```
:file:`/Users/{NAME}/python-basics`
```

Menu selections and GUI labels

1. *File* → *Save as* …
2. *Submit*

```
#. :menuselection:`File --> Save as ...`
#. :guilabel:`&Submit`
```

17.2.6 Lists

17.2.7 Numbered lists

1. First
2. Second
3. Third

```
#. First
#. Second
#. Third
```

Unnumbered lists

- Each entry in a list begins with an Asterisk (*).
- List items can be displayed for multiple lines as long as the list items remain indented.

```
* Each entry in a list begins with an Asterisk (``*``).
* List items can be displayed for multiple lines as long as the list items
  remain indented.
```

Definition lists

Term
: Definition of the term

Different term
: …and its definition

```
Term
    Definition of the term
Different term
    ...and its definition
```

17.2.8 Nested lists

- Lists can also be nested
  - and contain subitems

```
* Lists can also be nested

  * and contain subitems
```

17.2.9 Literal blocks

«Block quotation marks look like paragraphs, but are indented with one or more spaces.»

```
«Block quotation marks look like paragraphs, but are indented with one
or more spaces.»
```

17.2.10 Line blocks

Because of the pipe character, this becomes one line.
And this will be another line.

```
| Because of the pipe character, this becomes one line.
| And this will be another line.
```

17.2.11 Code blocks

Blocks of code are introduced and indented with a colon:

```
import docutils
print help(docutils)
```

```
>>> print 'But doctests start with ">>>" and don't need to be indented.'
```

```
Blocks of code are introduced and indented with a colon::

    import docutils
    print help(docutils)

>>> print 'But doctests start with ">>>" and don't need to be indented.'
```

See also:

Code blocks

17.2.12 Tables

Column heading	Column heading	Column heading	Column heading
body row 1, column 1	body row 1, column 2	body row 1, column 3	body row 1, column 4
body row 2, column 1	body row 2, column 2	body row 2, column 3, colspan 2	
body row 3, column 1	body row 3, column 2	body row 3, column 3, rowspan 2	body row 4, column 4
body row 5, column 1	body row 5, column 2		body row 5, column 4

```
+----------------+----------------+----------------+----------------+
| Column heading | Column heading | Column heading | Column heading |
+================+================+================+================+
| body row 1,    | body row 1,    | body row 1,    | body row 1,    |
| column 1       | column 2       | column 3       | column 4       |
+----------------+----------------+----------------+----------------+
| body row 2,    | body row 2,    | body row 2,                     |
| column 1       | column 2       | column 3,  colspan 2            |
+----------------+----------------+----------------+----------------+
| body row 3,    | body row 3,    | body row 3,    | body row 4,    |
| column 1       | column 2       | column 3,      | column 4       |
+----------------+----------------+ rowspan 2      +----------------+
| body row 5,    | body row 5,    |                | body row 5,    |
| column 1       | column 2       |                | column 4       |
+----------------+----------------+----------------+----------------+
```

17.2.13 Comments

```
.. A comment block begins with two points and can be indented further
```

17.3 Code blocks

Code blocks can be easily represented with the *code-block* directive. Together with Pygments, Sphinx will automatically highlight the syntax. You can specify the appropriate language for a code block with

.. code-block:: LANGUAGE

You can use this for example like this:

```
.. code-block:: python

   import this
```

Optionen

:linenos:

For *code-block*, the `linenos` option can also be used to specify that the code block should be displayed with line numbers:

```
.. code-block:: python
   :linenos:

   import this
```

```
1 import this
```

:lineno-start:

Die erste Zeilennummer kann mit der `lineno-start`-Option ausgewählt werden; `linenos` wird dann automatisch aktiviert: The first line number can be selected with the `lineno-start` option; `linenos` will then be activated automatically:

```
.. code-block:: python
   :lineno-start: 10

   import antigravity
```

```
10 import antigravity
```

:emphasize-lines:

`emphasize-lines` allows you to emphasise individual lines.

`.. literalinclude:: FILENAME`

allows you to include external files.

Options

:emphasize-lines:

:linenos:

Here is an example from our Jupyter Tutorial:

```
.. literalinclude:: main.py
   :emphasize-lines: 4, 9-12, 20-22
   :linenos:
```

```
1  from typing import Optional
2
3  from fastapi import FastAPI
4  from pydantic import BaseModel
5
6  app = FastAPI()
7
8
9  class Item(BaseModel):
10     name: str
```

(continues on next page)

(continued from previous page)

```
    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}
```

:diff:

If you want to show the diff of your code, you can specify the old file with the diff option, for example:

```
.. literalinclude:: main.py
   :diff: main.py.orig
```

```
--- /home/docs/checkouts/readthedocs.org/user_builds/python-basics-tutorial/
↪checkouts/24.1.0/docs/document/main.py.orig
+++ /home/docs/checkouts/readthedocs.org/user_builds/python-basics-tutorial/
↪checkouts/24.1.0/docs/document/main.py
@@ -1,8 +1,15 @@
 from typing import Optional

 from fastapi import FastAPI
+from pydantic import BaseModel

 app = FastAPI()
+
+
+class Item(BaseModel):
+    name: str
+    price: float
+    is_offer: Optional[bool] = None


 @app.get("/")
@@ -13,3 +20,8 @@
 @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):
```

(continues on next page)

(continued from previous page)

```
+    return {"item_name": item.name, "item_id": item_id}
```

17.3.1 Obsolete code

`.. deprecated:: version`

Describes when the function became obsolete. An explanation can also be given to inform what should be used instead. For example

```
.. deprecated:: 4.1
   instead use :func:`new_function`.
```

Deprecated since version 4.1: instead use `new_function()`.

`:py:module:deprecated:`

Marks a Python module as obsolete; it is then marked as such in various places.

17.4 Placeholder

Sphinx distinguishes the following placeholder variables:

`:envvar:`

Environment variable that also creates a reference to the appropriate `envvar` directive if it exists.

`:file:`

The name of a file or directory. Curly brackets can be used to specify a variable part, for example:

```
... is installed in :file:`/usr/lib/python3.{x}/site-packages` ...
```

In the generated HTML documentation, the `x` is specially marked with `em .pre` and italicised to show that it is to be replaced by the specific Python version.

`:makevar:`

The name of a **make** variable

`:samp:`

Text example, such as code within which curly braces can be used to indicate a variable part, as in *file* or in `print 1+`*VARIABLE*.

As of Sphinx1.8, curly braces can be displayed with a backslash (\).

Note:

`:content:`

This role has no special meaning by default. You can therefore use it for anything, for example also for variable names.

See also:

- Sphinx awesome sampdirective

17.5 UI elements and interactions

:guilabel:

Labels that are presented as part of an interactive user interface should be marked with *guilabel*. Any label used in the interface should be identified with this role, including labels for buttons, window titles, field names, menu and menu selection names, and even values in selection lists.

A keyboard shortcut for GUI labelling can be inserted with an ampersand (&); this will underline the following letter in the output.

Cancel is achieved, for example, with the following distinction:

```
:guilabel:`&Cancel`
```

Note: If you want to insert an ampersand, you can simply double it.

:kbd:

This represents a sequence of keystrokes. The form of the key sequence may depend on platform- or application-specific conventions. If there are no corresponding conventions, the names of modifier keys should be written out to improve accessibility. Also, do not reference a specific keyboard label.

You can achieve `Ctrl-s`, for example, with the following markup:

```
:kbd:`Ctrl-s`
```

:menuselection:

A menu selection should be marked with the `menuselection` role. This is used to mark a complete sequence, including submenu selections and selections of specific operations or any subsequences. The names of the individual selections should be separated by `-->`.

View → Cell Toolba r→ Slideshow is achieved, for example, with the following markup:

```
:menuselection:`View --> Cell Toolbar --> Slideshow`
```

menuselection, just like *guilabel*, also supports keyboard shortcuts with an ampersand (&).

17.6 Directives

reStructuredText can be expanded with Directives. Sphinx makes extensive use of this. Here are some examples:

17.6.1 Table of Contents

Docstrings

With the Sphinx extension sphinx.ext.autodoc, docstrings can also be included in the documentation. The following three directives can be specified:

.. automodule::
.. autoclass::
.. autoexception::

These document a module, a class or an exception using the docstring of the respective object.

Installation

`sphinx.ext.autodoc` is usually already specified in the Sphinx configuration file `docs/conf.py`:

```
extensions = [
    'sphinx.ext.autodoc',
    ...
]
```

If your package and its documentation are part of the same repository, they will always have the same relative position in the filesystem. In this case you can simply edit the Sphinx configuration for `sys.path` to indicate the relative path to the package, so:

```
sys.path.insert(0, os.path.abspath('..'))
import requests
```

If you have installed your Sphinx documentation in a virtual environment, you can also install your package there with:

```
$ python -m pip install my.package
```

or, if you want to develop the package further with:

```
$ python -m pip install -e https://github.com/veit/my.package.git
```

Examples

Here are some examples from the API documentation for the requests module:

```
Docstrings example
==================

Developer Interface
-------------------

.. module:: requests

Main Interface
--------------

.. autofunction:: head

Exceptions
----------

.. autoexception:: requests.RequestException

Request Sessions
----------------

.. autoclass:: Session
   :inherited-members:
```

This leads to the docstrings-example, generated from the following docstrings:

- requests.head
- requests.RequestException
- requests.Session

Note: You should follow these guidelines when writing docstrings:

- **PEP 8#comments**
- **PEP 257#specification**

---

sphinx-autodoc-typehints

With **PEP 484** a standard method for expressing types in Python code was introduced. This also allows types to be expressed differently in docstrings. The variant with types according to **PEP 484** has the advantage that type testers and IDEs can be used for static code analysis.

Python 3 type annotations:

```
def func(arg1: int, arg2: str) -> bool:
    """Summary line.

    Extended description of function.

    Args:
        arg1: Description of arg1
        arg2: Description of arg2

    Returns:
        Description of return value

    """
    return True
```

Types in Docstrings:

```
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Args:
        arg1 (int): Description of arg1
        arg2 (str): Description of arg2

    Returns:
        bool: Description of return value

    """
    return True
```

Note:

PEP 484#suggested-syntax-for-python-2-7-and-straddling-code are currently not supported by Sphinx and do not appear in the generated documentation.

sphinx.ext.napoleon

The sphinx extension sphinx.ext.napoleon allows you to define different sections in docstrings, including:

- `Attributes`
- `Example`
- `Keyword Arguments`
- `Methods`
- `Parameters`
- `Warning`
- `Yield`

There are two styles of docstrings in `sphinx.ext.napoleon`:

- Google
- NumPy

The main differences are that Google uses indentations and NumPy uses underscores:

Google:

```
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Args:
        arg1 (int): Description of arg1
        arg2 (str): Description of arg2

    Returns:
        bool: Description of return value

    """
    return True
```

NumPy:

```
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Parameters
    ----------
    arg1 : int
        Description of arg1
    arg2 : str
```

(continues on next page)

(continued from previous page)

```
        Description of arg2

    Returns
    -------
    bool
        Description of return value

    """
    return True
```

You can find the detailed configuration options in sphinxcontrib.napoleon.Config.

```
.. toctree::
   :maxdepth: 2

   start
   docstrings
```

Meta information

Section author: Veit Schiele <veit@cusy.io>

Code author: Veit Schiele <veit@cusy.io>

```
.. sectionauthor:: Veit Schiele <veit@cusy.io>
.. codeauthor:: Veit Schiele <veit@cusy.io>
```

Note: By default, this information is not included in the output until you set the configuration for `show_authors` to `True`.

See also

See also:

Sphinx Directives

```
.. seealso::
    `Sphinx Directives
    <https://www.sphinx-doc.org/en/master/usage/restructuredtext/directives.html>`_
```

Glossary

environment
: A structure where information about all documents under the root is saved, and used for cross-referencing. The environment is pickled after the parsing stage, so that successive runs only need to read and parse new and changed documents.

source directory
: The directory which, including its subdirectories, contains all source files for one Sphinx project.

```
.. glossary::

   environment
       A structure where information about all documents under the root is
       saved, and used for cross-referencing.  The environment is pickled
       after the parsing stage, so that successive runs only need to read
       and parse new and changed documents.

   source directory
       The directory which, including its subdirectories, contains all
       source files for one Sphinx project.
```

17.7 Intersphinx

sphinx.ext.intersphinx allows the linking of other project documentation.

17.7.1 Configuration

In `docs/conf.py` Intersphinx must be indicated as an extension:

```
extensions = [
    ...
    'sphinx.ext.intersphinx',
    ]
```

External Sphinx documentation can then be specified, e.g. with:

```
intersphinx_mapping = {
    'python': ('https://docs.python.org/3', None),
    'bokeh':  ('https://bokeh.pydata.org/en/latest/', None)
}
```

However, alternative files can also be specified for an inventory, for example:

```
intersphinx_mapping = {
    'python': ('https://docs.python.org/3', (None, 'python-inv.txt'),
    ...
}
```

17.7.2 Determine link targets

To determine the links available in an inventory, you can enter the following, for example:

```
$ python -m sphinx.ext.intersphinx https://docs.python.org/3/objects.inv
c:function
    PyAnySet_Check                          c-api/set.html#c.PyAnySet_Check
    PyAnySet_CheckExact                     c-api/set.html#c.PyAnySet_CheckExact
    PyArg_Parse                             c-api/arg.html#c.PyArg_Parse
...
```

17.7.3 Create a link

In order to link to https://docs.python.org/3/c-api/arg.html#c.PyArg_Parse, one of the following variants can be specified:

PyArg_Parse()

```
:c:func:`PyArg_Parse`
```

PyArg_Parse()

```
:c:func:`!PyArg_Parse`
```

Parsing arguments

```
:c:func:`Parsing arguments <PyArg_Parse>`
```

17.7.4 Custom links

You can also create your own `intersphinx` assignments, e.g. if `objects.inv` in Beautiful Soup has errors.

The error can be corrected with:

1. Installation of `sphobjinv`:

```
$ python -m pip install sphobjinv
```

2. Then the original file can be downloaded with:

```
$ cd docs/build/
$ mkdir _intersphinx
$ !$
$ curl -O https://www.crummy.com/software/BeautifulSoup/bs4/doc/objects.inv
$ mv objects.inv bs4_objects.inv
```

3. Change the Sphinx configuration `docs/conf.py`:

```
intersphinx_mapping = {
    ...
    'bs4':    ('https://www.crummy.com/software/BeautifulSoup/bs4/doc/', "_
↪intersphinx/bs4_objects.inv")
}
```

4. Convert to a text file:

```
$ sphobjinv convert plain bs4_objects.inv bs4_objects.txt
```

5. Editing the text file

   e.g.:

```
bs4.BeautifulSoup           py:class  1 index.html#beautifulsoup -
bs4.BeautifulSoup.get_text  py:method 1 index.html#get-text      -
bs4.element.Tag             py:class  1 index.html#tag           -
```

   These entries can then be referenced in a Sphinx documentation with:

```
- :class:`bs4.BeautifulSoup`
- :meth:`bs4.BeautifulSoup.get_text`
- :class:`bs4.element.Tag`
```

   See also:

   - Sphinx objects.inv v2 Syntax

6. Create a new `objects.inv` file:

```
$ sphobjinv convert zlib bs4_objects.txt bs4_objects.txt
```

7. Create Sphinx documentation:

```
$ python -m sphinx -ab html docs/ docs/_build/
```

17.7.5 Add roles

If you get an error message that a certain text role is unknown, e.g.

```
WARNING: Unknown interpreted text role "confval".
```

so you can add them in the `conf.py`:

```
def setup(app):
    # from sphinx.ext.autodoc import cut_lines
    # app.connect('autodoc-process-docstring', cut_lines(4, what=['module']))
    app.add_object_type(
        "confval",
        "confval",
        objname="configuration value",
        indextemplate="pair: %s; configuration value",
    )
```

17.8 Unified Modeling Language (UML)

17.8.1 Installation

1. Install plantuml:

```
$ sudo apt install plantuml
```

```
$ brew install plantuml
```

2. Install sphinxcontrib-plantuml:

```
$ bin/python -m pip install sphinxcontrib-plantuml
```

```
C:> Scripts\python -m pip install sphinxcontrib-plantuml
```

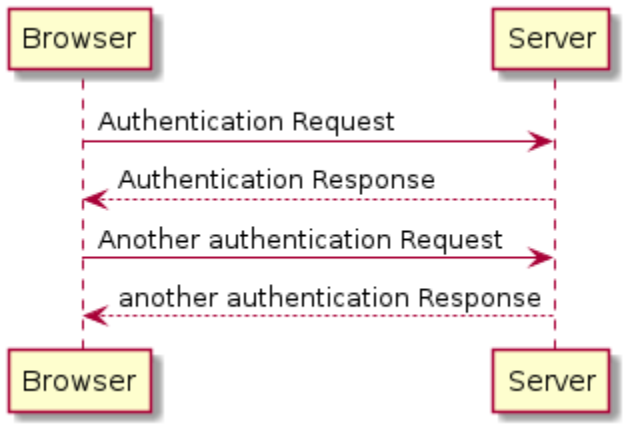
3. We then configure the `conf.py`:

```
extensions = [
    ...,
    'sphinxcontrib.plantuml',
    ]

plantuml = '/PATH/TO/PLANTUML'
```

Note: Also in Windows, the path is specified with `/`.

Sequence diagram




```
.. uml::

   Browser -> Server: Authentication Request
   Server --> Browser: Authentication Response
```

(continues on next page)

(continued from previous page)

```
   Browser -> Server: Another authentication Request
   Browser <-- Server: another authentication Response
```

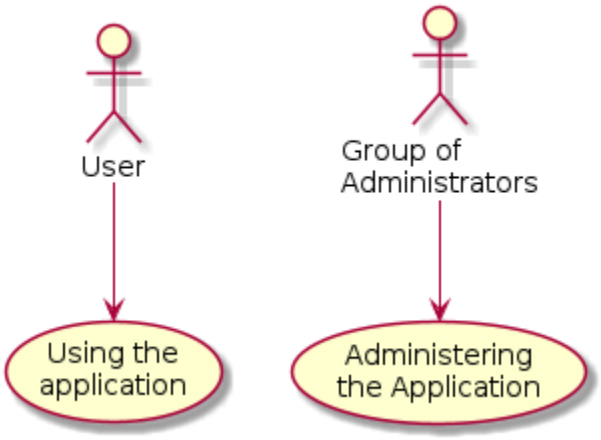
`->`
: is used to draw a message between two actors. The actors do not have to be declared explicitly.

`-->`
: is used to draw a dotted line.

<- and <--
: do not change the drawing, but may increase readability.

> **Note:** This applies only to sequence diagrams. In other diagrams other rules may apply.

Use Case diagram




```
.. uml::

   :User: --> (Use)
   "Group of\nAdministrators " as Admin
   "Using the\napplication" as (Use)
   Admin --> (Administering\nthe Application)
```

Use cases are enclosed by round brackets `()` and resemble an oval.

Alternatively, the `usecase` keyword can be used to define a use case. In addition, it is possible to define an alias using the `as` keyword. This alias can then be used when defining relationships.

You can add line breaks to the name of the use cases with `\n`.

Activity diagram

(*)
: Start and end nodes of the activity diagram.

  (*top)
  : In some cases, this can be used to move the start point to the beginning of the diagram.

-->
: defines an activity

  -down->
  : Down arrow (default value)

  -right-> or ->
  : Right arrow

  -left->
  : Arrow to the left

  -up->
  : Arrow up

if, then, else
: Keywords for the definition of branches.

  Example:

```
.. uml::

   (*) --> "Initialisation"
   if "a test" then
   -->[true] "An activity"
   --> "Another activity"
   -right-> (*)
   else
   ->[false] "Something else"
   -->[end the processes] (*)
   endif
```

fork, fork again and end fork or end merge
: Keywords for parallel processing.

  Example:

```
.. uml::

   start
   fork
     :action 1;
   fork again
     :action 2;
   end fork
   stop
```

Class diagram

`abstract class`, `abstract`

Example:

```
.. uml::

   abstract class  "Abstract class"
```

`annotation`

```
.. uml::

   annotation      Annotation
```

`circle`, `()`

```
.. uml::

   circle          Circle
```

`class`

```
.. uml::

   class           Class
```

`diamond`, `<>`

An empty diamond stands for an association, a black diamond for a composition.

```
.. uml::

   diamond         Association
```

`entity`

```
.. uml::

   entity          Entity
```

`enum`

```
.. uml::

   enum            Enumeration
```

`interface`

```
.. uml::

   interface       Interface
```

17.9 Extensions

See also:

Sphinx Extensions

17.9.1 Built-in extensions

sphinx.ext.autodoc
Integrate documentation from docstrings

sphinx.ext.autosummary
generates summaries of functions, methods and attributes from docstrings

sphinx.ext.autosectionlabel
references section using the title

sphinx.ext.graphviz
Rendering of Graphviz graphs

sphinx.ext.ifconfig
includes content only under certain conditions

sphinx.ext.intersphinx
allows the linking of other project documentation

sphinx.ext.mathjax
Rendering via JavaScript

sphinx.ext.napoleon
Support for NumPy and Google style docstrings

sphinx.ext.todo
Support for ToDo items

sphinx.ext.viewcode
adds links to the source code of the Sphinx documentation

See also:

sphinx/sphinx/ext/

17.9.2 Third-party extensions

nbsphinx
Jupyter Notebooks in Sphinx

jupyter-sphinx
allows rendering of Jupyter interactive widgets in Sphinx.

See also:

Embedding Widgets in the Sphinx HTML Documentation

Breathe
ReStructuredText and Sphinx bridge to Doxygen

numpydoc
NumPy's Sphinx extension

Releases
: writes a changelog file

sphinxcontrib-napoleon
: Napoleon is a pre-processor for parsing NumPy- and Google-style docstrings

sphinx-autodoc-annotation
: use Python 3 annotations in sphinx-enabled docstrings

sphinx-autodoc-typehints
: Type hints support for the Sphinx autodoc extension

sphinx-git
: git-Changelog for Sphinx

Sphinx Gitstamp Generator Extension
: inserts a git datestamp into the context

sphinx-intl
: Sphinx extension for translations

sphinx-autobuild
: monitors a Sphinx repository and creates new documentation as soon as changes are made

Sphinx-Needs
: allows the definition, linking and filtering of need-objects, for example requirements and test cases

Sphinx-pyreverse
: generate a UML diagram from python modules

sphinx-jsonschema
: display a JSON Schema in the Sphinx documentation

Sphinxcontrib-mermaid
: allows you to embed Mermaid graphics in your documents.

Sphinx Sitemap Generator Extension
: generate multiversion and multilanguage sitemaps for the HTML version

Sphinx Lint
: based on rstlint.py from CPython.

sphinx-toolbox
: Toolbox for Sphinx with many useful tools.

See also:

sphinx-contrib
: A repository of Sphinx extensions maintained by their respective authors.

sphinx-extensions
: Curated site with Sphinx extensions with live examples and their configuration.

17.9.3 Own Extensions

Local extensions in a project should be specified relative to the documentation. The appropriate path is specified in the Sphinx configuration `docs/conf.py`. If your extension is in the directory `exts` in the file `foo.py`, then the `conf.py` should look like this:

```
import sys
import os
sys.path.insert(0, os.path.abspath('exts'))

extensions = [
    'foo',
    ...
    ]
```

See also:

- Developing extensions for Sphinx
- Application API

17.10 Testing

17.10.1 Build error

You have the option of checking that your content is built correctly before publishing your changes. For this purpose, Sphinx has a nitpicky mode that can be invoked with the `-n` option, for example with:

```
$ bin/python -m sphinx -nb html docs/ docs/_build/
```

```
C:> Scripts\python -m sphinx -nb html docs\ docs\_build\
```

17.10.2 Check links

You can also automatically ensure that the link targets you specify are accessible. Sphinx uses a linkcheck builder for this purpose, which you can call with:

```
$ bin/python -m sphinx -b linkcheck docs/ docs/_build/
```

```
C:> Scripts\python -m sphinx -b linkcheck docs\ docs\_build\
```

The output may then look like this:

```
$ bin/python -m sphinx -b linkcheck docs/ docs/_build/
Running Sphinx v3.5.2
loading translations [de]... done
...
building [mo]: targets for 0 po files that are out of date
building [linkcheck]: targets for 27 source files that are out of date
...
(content/accessibility: line   89) ok        https://bbc.github.io/subtitle-guidelines/
```

(continues on next page)

(continued from previous page)

```
(content/writing-style: line  164) ok        http://disabilityinkidlit.com/2016/07/08/
↪introduction-to-disability-terminology/

...
(   index: line    5) redirect  https://cusy-design-system.readthedocs.io/ - with Found␣
↪to https://cusy-design-system.readthedocs.io/de/latest/
...
(accessibility/color: line  114) broken    https://chrome.google.com/webstore/detail/
↪nocoffee/jjeeggmbnhckmgdhmgdckeigabjfbddl - 404 Client Error: Not Found for url:␣
↪https://chrome.google.com/webstore/detail/nocoffee/jjeeggmbnhckmgdhmgdckeigabjfbddl
```

```
C:> Scripts\python -m sphinx -b linkcheck docs\ docs\_build\
Running Sphinx v3.5.2
loading translations [de]... done
...
building [mo]: targets for 0 po files that are out of date
building [linkcheck]: targets for 27 source files that are out of date
...
(content/accessibility: line   89) ok        https://bbc.github.io/subtitle-guidelines/
(content/writing-style: line  164) ok        http://disabilityinkidlit.com/2016/07/08/
↪introduction-to-disability-terminology/

...
(   index: line    5) redirect  https://cusy-design-system.readthedocs.io/ - with Found␣
↪to https://cusy-design-system.readthedocs.io/de/latest/
...
(accessibility/color: line  114) broken    https://chrome.google.com/webstore/detail/
↪nocoffee/jjeeggmbnhckmgdhmgdckeigabjfbddl - 404 Client Error: Not Found for url:␣
↪https://chrome.google.com/webstore/detail/nocoffee/jjeeggmbnhckmgdhmgdckeigabjfbddl
```

Code formatting

blacken-docs currently supports the following black options:

- `-l/--line-length`
- `-t/--target-version`
- `-s/--skip-string-normalization`
- `-E/--skip-errors`

```
$ bin/python -m pip install blacken-docs
```

17.11 shot-scraper

shot-scraper is a tool to automate the process of updating screenshots.

17.11.1 Installation

```
$ python -m pip install shot-scraper
$ shot-scraper install
```

Note: The second line installs the required browser.

17.11.2 Use

shot-scraper can be used in two ways

1. …for single screenshots on the command line:

   ```
   $  shot-scraper https://jupyter-tutorial.readthedocs.io/de/latest/clean-prep/index.
   ↪html -o ~/Downloads/clean-prep.png
   ```

   …or with additional options, e.g. for JavaScript and CSS selectors:

   ```
   $ shot-scraper https://jupyter-tutorial.readthedocs.io/de/latest/clean-prep/
   ↪index.html -s '#overview' -o ~/Downloads/clean-prep.png
   ```

2. …for a set of screenshots configured in a YAML file:

   ```
   - url: https://jupyter-tutorial.readthedocs.io/de/latest/clean-prep/index.html
     output: ~/Downloads/clean-prep.png
   - url: https://www.example.org/
     width: 736
     quality: 40
     output: example.jpg
   ```

   Afterwards `shot-scraper multi` can be used, for example:

   ```
   $ shot-scraper multi shots.yaml
   Screenshot of 'https://jupyter-tutorial.readthedocs.io/de/latest/clean-prep/index.
   ↪html' written to '~(Downloads/clean-prep.png'
   Screenshot of 'https://www.example.org/' written to 'example.jpg'
   ```

   See also:

   - In the README.md file you will find a complete overview of the possible options.
   - In the shot-scraper-demo repository you will find a much more comprehensive shots.yaml file.

17.11.3 GitHub Actions

shot-scraper can be easily integrated into GitHub Actions. The shot-scraper-demo repository also contains an exemplary shots.yml. Once a day, two screenshots are created and transferred back to the repository. Note, however, that saving image files that change frequently can make the revision history very unreadable. Therefore, you should use shot-scraper with caution together with GitHub Actions.

17.12 Badges

Some of this information and more can be accessed as badges. They are helpful in getting a quick overview of a product. For the cookiecutter-namespace-template these are, for example:

You can also create your own badges, for example:

See also:

- shields.io

17.13 Sphinx

For extensive documentation you can, for example, use Sphinx, a documentation tool that converts reStructuredText into HTML or PDF, EPub and man pages. The Python Basics are also created with Sphinx. To get a first impression of Sphinx, you can have a look at the source code of this page by following the link Sources.

Originally, Sphinx was developed for the documentation of Python and is now used in almost all Python projects, including NumPy and SciPy, Matplotlib, Pandas and SQLAlchemy.

The Sphinx autodoc feature, which can be used to create documentation from Python *Docstrings*, may also be conducive to the spread of Sphinx among Python developers. Overall, Sphinx allows developers to create complete documentation in place. Often the documentation is also stored in the same Git repository, so that the creation of the latest software documentation remains easy.

Sphinx is also used in projects outside the Python community, e.g. for the documentation of the Linux kernel: Kernel documentation update.

Read the Docs was developed to forther simplify documentation. Read the Docs makes it easy to create and publish documentation after each commit.

For project documentation, visualising Git feature branches and tags with git-big-picture can be helpful.

Note: If the content of `long_description` in `setup()` is written in reStructured Text, it is displayed as well-formatted HTML on ther *Python Package Index* (*PyPI*).

17.14 Other documentation tools

Pycco
: is a Python port of Docco.

CHAPTER

EIGHTEEN

APPENDIX

18.1 Regular expressions

See also:

- www.regular-expressions.info
- AutoRegex

18.1.1 []

Square brackets define a list or range of characters to search for:

[abc]
corresponds to a, b or c

[a-z]
corresponds to any lower case letter

[A-Za-z]
corresponds to each letter

[A-Za-z0-9]
corresponds to any letter or digit

18.1.2 Number

.
corresponds to a single character

*****
corresponds to zero or more times the preceding element, for example `colou*r` matches `color`, `colour`, `colouur` etc.

?
corresponds to zero or once the preceding element. `colou?r` matches `color` and `colour`.

+
matches the previous element one or more times, for example `.+` matches `.`, `..`, `...` etc.

{N}
corresponds `N` times to the preceding element.

{N,}
matches the previous element `N` or more times.

{N,M}
corresponds at least N times to the preceding element, but not more than M times.

18.1.3 Position

^
puts the position at the beginning of the line.

$
puts the position at the end of the line.

18.1.4 Link

|
means *or*.

18.1.5 Escape characters and literals

****
is used to search for a special character, for example to find `.org` you have to use the regular expression `\.org` because `.` is the special character that matches every character.

\d
matches every single digit.

\w
matches any part of a word character and is equivalent to `[A-Za-z0-9]`.

\s
matches any space, tab or newline.

\b
matches a pattern on a word boundary.

18.2 Unicode and character encodings

There are dozens of character encodings. For an overview of Python's encodings, see Encodings and Unicode.

18.2.1 The `string` module

Python's string module distinguishes the following string constants, all of which fall into the ASCII character set:

```
# Some strings for ctype-style character classification
whitespace = ' \t\n\r\v\f'
ascii_lowercase = 'abcdefghijklmnopqrstuvwxyz'
ascii_uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
ascii_letters = ascii_lowercase + ascii_uppercase
digits = '0123456789'
hexdigits = digits + 'abcdef' + 'ABCDEF'
octdigits = '01234567'
```

(continues on next page)

(continued from previous page)

```
punctuation = r"""!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"""
printable = digits + ascii_letters + punctuation + whitespace
```

Most of these constants should be self-explanatory in their identifier names. `hexdigits` and `octdigits` refer to the hexadecimal and octal values respectively. You can use these constants for everyday string manipulation:

```
>>> import string
>>> hepy = "Hello Pythonistas!"
>>> hepy.rstrip(string.punctuation)
'Hello Pythonistas'
```

However, the string module works with Unicode by default, which is represented as binary data (bytes).

18.2.2 Unicode

It is obvious that the ASCII character set is not nearly large enough to cover all languages, dialects, symbols and glyphs; it is not even large enough for English.

While ASCII is a complete subset of Unicode – the first 128 characters in the Unicode table correspond exactly to ASCII characters – Unicode encompasses a much larger set of characters. Unicode itself is not an encoding but is implemented by various character encodings, with UTF-8 probably being the most commonly used encoding scheme.

Note: The Python help documentation has an entry for Unicode: enter `help()` and then `UNICODE`. The various options for creating Python strings are described in detail.

See also:

- Unicode HOWTO
- What's New In Python 3.0: Text Vs. Data Instead Of Unicode Vs. 8-bit

Unicode and UTF-8

While Unicode is an abstract encoding standard, UTF-8 is a concrete encoding scheme. The Unicode standard is a mapping of characters to code points and defines several different encodings from a single character set. UTF-8 is an encoding scheme for representing Unicode characters as binary data with one or more bytes per character.

18.2.3 Encoding and decoding in Python 3

The str type is intended for the representation of human-readable text and can contain all Unicode characters. The bytes type, on the other hand, represents binary data that is not inherently encoded. `str.encode()` and `bytes.decode()` are the methods of transition from one to the other:

```
>>> "You're welcome!".encode("utf-8")
b'You\xe2\x80\x99re welcome!'
>>> b"You\xe2\x80\x99re welcome!".decode("utf-8")
'You're welcome!'
```

The result of `str.encode()` is a bytes object. Both byte literals (such as `b'You\xe2\x80\x99re welcome!'`) and representations of bytes only allow ASCII characters. For this reason, when calling `"You're welcome!".encode("utf-8")`, the ASCII-compatible `'You'` may be represented as it is, but the ' becomes `'\xe2\x80\x99'`. This chaotic looking sequence represents three bytes, `e2`, `80` and `99` as hexadecimal values.

Tip: In `.encode()` and `.decode()`, the encoding parameter is `"utf-8"` by default; however, it is recommended to specify it explicitly.

With `bytes.fromhex()` you can convert the hexadecimal values into bytes:

```
>>> bytes.fromhex('e2 80 99')
b'\xe2\x80\x99'
```

UTF-16 and UTF-32

The difference between these and UTF-8 is considerable in practice. In the following, I would like to show you only briefly by means of an example that a round-trip conversion can simply fail here:

```
>>> hepy = "Hello Pythonistas!"
>>> hepy.encode("utf-8")
b'Hello Pythonistas!'
>>> len(hepy.encode("utf-8"))
18
>>> hepy.encode("utf-8").decode("utf-16")
'\u206f'
>>> len(hepy.encode("utf-8").decode("utf-16"))
9
```

Encoding Latin letters in UTF-8 and then decoding them in UTF-16 resulted in a text that also contains characters from the Chinese, Japanese or Korean language areas as well as Roman numerals. Decoding the same byte object can lead to results that are not even in the same language or contain the same number of characters.

18.2.4 Python 3 and Unicode

Python 3 relies fully on Unicode and specifically on UTF-8:

- Python 3 source code is assumed to be UTF-8 by default.
- Texts (str) are Unicode by default. Encoded Unicode text is represented as binary data (Bytes) dargestellt.
- Python 3 accepts many Unicode code points in identifiers.
- Python's re module uses the `re.UNICODE` flag by default, not `re.ASCII`. This means that, for example, `r"\w"` matches Unicode word characters, not just ASCII letters.
- The default encoding in `str.encode()` and `bytes.decode()` is UTF-8.

The only exception could be `open()`, which is platform dependent and therefore depends on the value of `locale.getpreferredencoding()`:

```
>>> import locale
>>> locale.getpreferredencoding()
'UTF-8'
```

18.2.5 Built-in Python Functions

Python has a number of built-in functions that relate to character encodings in some way:

`ascii()`, `bin()`, `hex()`, `oct()`
: output a string.

`bytes`, `str`, `int`
: are class constructors for their respective types, converting the input to the desired type.

`ord()`, `chr()`
: are inverses of each other in that the Python function `ord()` converts an `str` character to its `base=10` code point, while `chr()` does the opposite.

Below is a more detailed look at each of these nine functions:

Function	RReturn type	Description
`ascii()`	`str`	ASCII representation of an object, escaping non-ASCII characters.
`bin()`	`str`	binary representation of an integer with the prefix `0b`
`hex()`	`str`	hexadecimal representation of an integer with the prefix `0x`
`oct()`	`str`	octal representation of an integer with the prefix `0o`
`bytes`	`bytes`	converts the input to bytes type
`str`	`str`	converts the input to str type
`int`	`int`	converts the input to `int type`
`ord()`	`int`	converts a single Unicode character to its integer code point
`chr()`	`str`	converts an integer code point into a single Unicode character

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M