Python Basics

Release 24.1.0

Veit Schiele

03.05.2024

Inhaltsverzeichnis

Willkommen bei den Python Basics! Ich habe dieses Buch geschrieben, um einen einfachen und praxisnahen Einstieg in Python zu ermöglichen. Das Buch ist nicht als umfassendes Nachschlagewerk für Python gedacht, sondern das Ziel ist vielmehr, euch grundlegend mit Python vertraut zu machen und euch schnell das Schreiben eigener Programme zu ermöglichen.

Bemerkung: Wenn ihr Vorschläge für Verbesserungen und Ergänzungen habt, freue ich mich über eure Verbesserungsvorschläge.

KAPITEL 1

Einführung

1.1 Über Python

Vielleicht stellt ihr euch die Frage, warum ihr Python lernen solltet. Es gibt viele Programmiersprachen von C und C++ über Java bis hin zu Lua und Go.

Abb. 1: TIOBE Index für Oktober 2022

Python hat eine sehr große Verbreitung gefunden und einer der Gründe dürfte sein, dass sie auf vielen verschiedenen Plattformen läuft, von IoT-Geräten über die gängigen Betriebssysteme bis hin zu Supercomputern. Es kann gut zur Entwicklung kleiner Anwendungen und schneller Prototypen verwendet werden. Dabei gibt es unzählige Software-Bibliotheken, die euch die Arbeit erleichtern.

Python ist eine moderne Programmiersprache, die von Guido van Rossum in den 90er Jahren entwickelt wurde.

Siehe auch:

- Lambert Meertens: The Origins of Python

Einige Stärken von Python sind:

leichte Nutzbarkeit
: Einige der Gründe hierfür sind, dass Typen mit Objekten verbunden sind, nicht mit Variablen; einer Variablen können Werte eines beliebigen Typs zugewiesen werden und eine Liste kann Objekte verschiedener Typen enthalten. Zudem sind die Syntaxregeln sehr einfach und ihr könnt schnell lernen, nützlichen Code zu schreiben.

Ausdrucksstärke
: Häufig könnt ihr in wenigen Zeilen Code sehr viel mehr erreichen als in anderen Sprachen. Dies führt dazu, dass ihr eure Projekte schneller abschließen könnt und auch Fehlersuche und Wartung sehr vereinfacht werden.

Lesbarkeit
: Die leichte Lesbarkeit von Python-Code vereinfacht die Fehlersuche und Wartung. Dies erreicht Python U.A. (unter anderem) durch erforderliche Einrückungen.

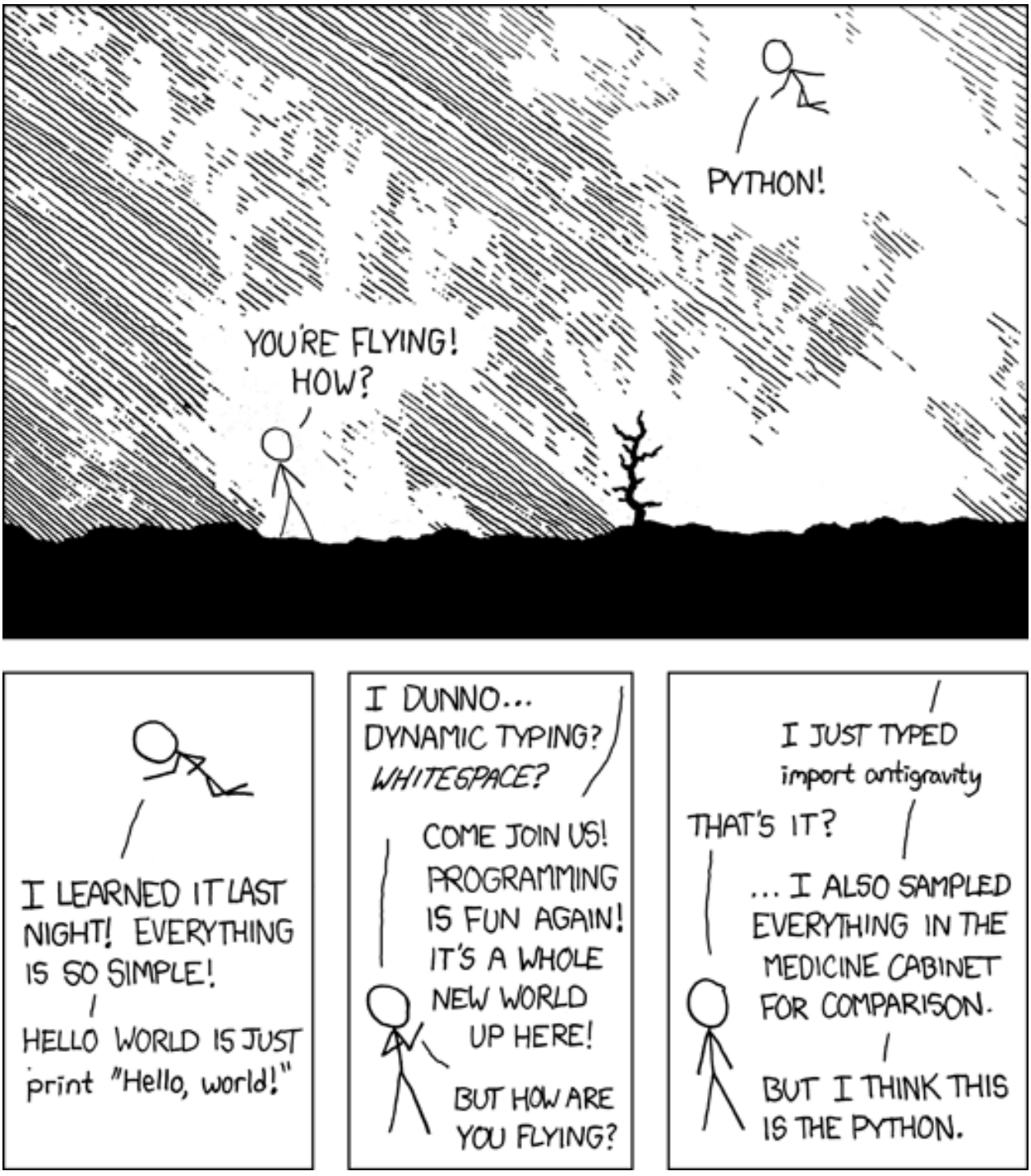


Abb. 2: XKCD: Python

Vollständigkeit
: Mit der Installation von Python ist bereits alles wesentliche vorhanden, was für das Programmieren mit Python benötigt wird, E-Mails, Websites, Datenbanken, ohne dass zusätzliche Bibliotheken installiert werden müssen.

Plattformunabhängigkeit
: Python läuft auf vielen Plattformen: Windows, Mac, Linux etc. Es gibt sogar Varianten, die auf Java (Jython) und .NET (IronPython) laufen.

Open Source
: Ihr könnt Python herunterladen und für die Entwicklung kommerzieller oder privater Anwendungen frei verwenden. Dabei wird Python von vielen etablierten Unternehmen genutzt und gefördert, U.A. von Google, Meta und Bloomberg. Und wenn ihr etwas zurückgeben wollt, könnt ihr dies ebenfalls gerne machen : Python Software Foundation Sponsorship.

Python hat zwar einige Vorteile, aber keine Sprache ist in allen Bereichen die beste Lösung. So schneidet z.B. (zum Beispiel) Python in den folgenden Bereichen weniger gut ab:

Geschwindigkeit
: Python ist keine vollständig kompilierte Sprache und der Code wird zunächst in Bytecode kompiliert bevor er vom Python-Interpreter ausgeführt wird. Zwar gibt es einige Aufgaben, wie z.B. das Parsen von Zeichenketten mit regulären Ausdrücken, für die Python effiziente Implementierungen bereitstellt, und die genauso schnell wie ein C-Programm sind, dennoch werden Python-Programme in den meisten Fällen langsamer sein als C-Programme. Dies spielt jedoch selten eine entscheidende Rolle, da es bereits viele Python-Module gibt, die intern C verwenden.

  Siehe auch:

  - Performance

Bibliotheksvielfalt
: Python verfügt zwar bereits über sehr viele Bibliotheken, in einigen Fällen werdet ihr jedoch passende Bibliotheken nur in anderen Sprachen finden. Für die meisten Probleme, die programmatisch gelöst werden sollen, ist die Bibliotheksunterstützung von Python jedoch hervorragend.

Variablentypen
: Anders als in vielen anderen Sprachen sind Variablen keine Container, sondern eher Etiketten, die auf verschiedene Objekte verweisen: Ganzzahlen, Zeichenketten, Klasseninstanzen und vieles mehr. Manche empfinden es als Nachteil, dass Python hier nicht einfach eine Typvalidierung durchführt, aber die Anzahl der Typfehler ist meist überschaubar und die Flexibilität der dynamischen Typisierung wiegt die Probleme meist auf.

Unterstützung für mobile Geräte
: Auch wenn in den letzten Jahren die Anzahl der mobilen Geräte stark zugenommen hat, so ist Python in diesem Bereich doch nicht stark vertreten. Es gibt zwar ein paar Optionen, Python auf mobile Geräte zu verteilen und auszuführen, dies ist jedoch nicht immer einfach.

Unterstützung für nebenläufige Berechnungen
: Prozessoren mit mehreren Kernen sind inzwischen weit verbreitet und führen in vielen Bereichen zu erheblichen Leistungssteigerungen. Die Standardimplementierung von Python ist jedoch nicht für die Nutzung mehrerer Kerne ausgelegt.

  Siehe auch:

  - Einführung in Multithreading, Multiprocessing und async

KAPITEL 2

Installation

Die Installation von Python ist einfach. Der erste Schritt besteht darin, die aktuelle Version von www.python.org/downloads herunterzuladen. Das Tutorial basiert auf Python 3.10, falls ihr jedoch Python 3.7 oder 3.8 installiert habt, ist das auch kein Problem.

Bei den meisten Linux-Distributionen ist Python bereits installiert. Wenn eine vorkompilierte Version von Python in eurer Linux-Distribution existiert, empfehle ich euch, diese zu verwenden.

Wenn ihr dennoch eine aktuellere Python-Version installieren wollt, könnt ihr dies z.B. für Debian oder Ubuntu wie folgt tun:

```
$ wget https://www.python.org/ftp/python/3.12.3/Python-3.12.3.tgz
$ tar xf Python-3.12.3.tgz
$ cd Python-3.12.3
$ ./configure --enable-optimizations
$ sudo make altinstall
```

Ihr benötigt eine Python-Version, die zu eurem macOS und eurem Prozessor passt. Wenn ihr die richtige Variante ermittelt habt, könnt ihr die Image-Datei herunterladen und mit einem Doppelklick mounten und anschließend das darin enthaltene Installationsprogramm starten. Anschließend befindet sich Python im `Programme`-Ordner.

Wenn ihr Homebrew verwendet, könnt ihr Python auch einfach im Terminal installieren mit:

```
$ brew install python3
```

Python kann für die meisten Windows-Versionen nach Windows 7 mit dem Python-Installationsprogramm in drei Schritten installiert werden:

1. Ladet das aktuelle Installationsprogramm von Python Releases for Windows herunter, z.B. Windows installer (64-bit).
2. Startet das Installationsprogramm. Sofern ihr die notwendigen Berechtigungen habt, installiert Python mit der Option *Install launcher for all users*. Dies sollte Python in `C:\Program Files\Python310-64` installieren. Außerdem sollte *Add Python 3.10 to PATH* aktiviert sein damit dieser Pfad zur Python-Installation auch in der Liste der `PATH`-Umgebungsvariablen eingetragen wird.
3. Schließlich könnt ihr die Installation nun überprüfen, indem ihr in der Eingabeaufforderung folgendes eingebt:

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

Bemerkung: Wenn auf eurem System bereits Python installiert ist, könnt ihr problemlos euer eigenes Python installieren. Eine neue Version ersetzt nicht die alte sondern wird an einem anderen Ort installiert.

KAPITEL 3

Editoren

3.1 Interaktive Shell

Mit der interaktiven Shell könnt ihr einfach die meisten Beispiele in diesem Tutorial ausführen. Später lernt ihr auch, wie ihr Code, der in eine Datei geschrieben wurde, einfach als Modul eingebunden werden kann.

Gebt `python3` im Terminal ein:

```
$ 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.
>>>
```

Öffnet ein Terminal-Fenster und gebt dort `python3` ein:

```
$ 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.
>>>
```

Bemerkung: Wenn ihr die Fehlermeldung *Kommando nicht gefunden.* erhaltet, könnt ihr `Update Shell Profile` ausführen, das sich in `Programme/Python3.10` befindet.

Die interaktive Python-Shell könnt ihr starten in *Start* → *Programme* → *Python 3.10*

Alternativ könnt ihr auch die direkt ausführbare Datei `Python.exe` suchen, z.B. in `C:\Users\VEIT\AppData\Local\Programme\Python\Python310-64` und dann doppelklicken.

Mit den Pfeiltasten, `Home`, `End`, `Page up` und `Page down` könnt ihr durch frühere Eingaben blättern und mit der Eingabetaste wiederholen.

3.1.1 Beenden der interaktiven Shell

Um die interaktive Shell zu beenden, könnt ihr einfach `Ctrl-d` unter Linux und macOS verwenden oder `Ctrl-z` unter Windows. Alternativ könnt ihr auch `exit()` eingeben.

3.2 IDLE

IDLE ist das Akronym für eine integrierte Entwicklungsumgebung (engl.: integrated development environment) und kombiniert einen interaktiven Interpreter mit Werkzeugen zur Code-Bearbeitung und Fehlersuche. Das Ausführen ist sehr einfach auf den verschiedenen Plattformen:

Gebt folgendes in eurem Terminal ein:

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

Ihr könnt IDLE starten in *Windows → Alle Apps → IDLE (Python GUI)*

Ihr könnt mit den Tasten `alt-p` und `alt-n` durch die Historie der vorherigen Befehle blättern.

KAPITEL 4

Python erkunden

Egal, ob ihr IDLE oder die interaktive Shell nutzt, es gibt einige nützliche Funktionen, um Python zu erkunden.

4.1 help()

`help()` hat zwei verschiedene Modi. Wenn ihr `help()` eingebt, ruft ihr das Hilfesystem auf, mit dem ihr Informationen zu Modulen, Schlüsselwörtern und weiteren Themen erhalten könnt. Wenn ihr euch im Hilfesystem befindet, seht ihr mit `help>` eine Eingabeaufforderung. Ihr könnt nun einen Modulnamen eingeben, z.B. `float`, um die Python-Dokumentation zu diesem Typ zu durchsuchen.

`help()` ist Teil der pydoc-Bibliothek, die Zugriff auf die in Python-Bibliotheken integrierte Dokumentation bietet. Da jede Python-Installation mit einer vollständigen Dokumentation ausgeliefert wird, habt ihr auch offline die gesamte Dokumentation zur Hand.

Alternativ könnt ihr `help()` auch gezielter anwenden, indem ihr einen Typ- oder Variablennamen als Parameter übergebt, z.B.:

```
>>> 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() und locals()

`dir()` ist eine weitere nützliche Funktion, die Objekte in einem bestimmten *Namensraum* auflistet. Wenn ihr sie ohne Parameter verwendet, könnt ihr herausfinden, welche Methoden und Daten lokal verfügbar sind. Alternativ kann sie auch Objekte für ein Modul oder einen Typ auflisten.

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

Im Gegensatz zu `dir()` zeigen sowohl `globals()` als auch `locals()` die mit den Objekten verbundenen Werte an. Aktuell geben beide Funktionen dasselbe zurück:

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

KAPITEL 5

Stil

5.1 Einrückung und Blöcke

Python unterscheidet sich von den meisten anderen Programmiersprachen, weil es Einrückungen verwendet, um die Struktur zu bestimmen (D.H. (das heißt) um zu bestimmen, was die *while*-Klausel einer Bedingung USW. (und so weiter) darstellt). Die meisten anderen Sprachen verwenden dazu geschweifte Klammern. Im folgenden Beispiel wird durch die Einrückung der Zeilen 3–6 festgelegt, dass sie zur `while`-Anweisung gehören:

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

Einrückungen zur Strukturierung des Codes anstelle von geschweiften Klammern ist zwar etwas gewöhnungsbedürftig, bietet aber erhebliche Vorteile:

- Ihr könnt weder fehlende oder zu viele Klammern haben. Auch müsst ihr nicht mehr nach der Klammer suchen, die zu früheren Klammern passen könnte.
- Die visuelle Struktur des Codes spiegelt seine tatsächliche Struktur wider, wodurch die Struktur des Codes sehr viel einfacher zu verstehen ist.
- Python Codierung-Styles sind meist einheitlich; M.A.W. (mit anderen Worten), euer Code wird meist sehr ähnlich aussehen, wie derjenige von anderen.

5.2 Kommentare

Meist ist alles, was hinter # folgt ein Kommentar und wird bei der Ausführung des Codes nicht beachtet. Die offensichtliche Ausnahme ist # in einer *Zeichenkette*:

```
>>> x = "# Dies ist eine Zeichenkette und kein Kommentar"
```

5.3 Grundlegender Python-Stil

In Python gibt es nur relativ wenige Einschränkungen für den Codierungsstil, mit der offensichtlichen Ausnahme, dass der Code durch Einrücken in Blöcke gegliedert werden muss. Selbst in diesem Fall ist nicht vorgeschrieben, wie (Tabulatoren oder Leerzeichen) und wie weit eingerückt wird. Es gibt jedoch bevorzugte stilistische Konventionen für Python, die im *Python Enhancement Proposal* (PEP) 8 enthalten sind. Eine Auswahl von Python-Konventionen findet ihr in der folgenden Tabelle:

Kontext	Empfehlung	Beispiel
Modul- und Paketnamen	kurz, Kleinbuchstaben, Unterstriche nur bei Bedarf	`math`, `sys`
Funktionsnamen	Kleinbuchstaben, GGF. (gegebenenfalls)` mit Unterstrichen	`my_func()`
Variablennamen	Kleinbuchstaben, GGF. (gegebenenfalls)` mit Unterstrichen	`my_var`
Klassennamen	CamelCase-Schreibweise	`MyClass`
Konstantennamen	Versalien mit Unterstrichen	`PI`
Einrückung	Vier Leerzeichen pro Ebene, keine Tabs	
Vergleiche	nicht explizit mit `True` oder `False`,siehe auch *Boolesche Werte und Ausdrücke*	`if my_var:`, `if not my_var:`

Siehe auch:

- **PEP 8**
- Google Python Style Guide

Ich empfehle dringend, die Konventionen von PEP 8 zu befolgen. Sie sind bewährt, und machen euren Code für euch selbst und andere leichter verständlich.

KAPITEL 6

Variablen und Ausdrücke

6.1 Variablen

Der am häufigsten verwendete Befehl in Python ist die Zuweisung. Der Python-Code um eine Vairiable namens `x` zu erstellen, die den Wert erhalten soll, lautet:

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

In Python ist, anders als in vielen anderen Programmiersprachen, weder eine Variablendeklaration noch ein Zeilenendebegrenzer notwendig. Die Zeile wird durch das Ende der Zeile abgeschlossen. Variablen werden automatisch erstellt, wenn sie zum ersten Mal zugewiesen werden.

Bemerkung: In Python sind Variablen Label, die auf Objekte verweisen. Eine beliebige Anzahl von Label können sich auf dasselbe Objekt beziehen, und wenn sich dieses Objekt ändert, ändert sich auch der Wert, auf den sich all diese Variablen beziehen. Um besser zu verstehen, was das bedeutet, seht euch das folgende Beispiel an:

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

Variablen können sich jedoch auch auf Konstanten beziehen:

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

In diesem Fall verweisen nach der dritten Zeile `x`, `y` und `z` alle auf dasselbe unveränderlichee Integer-Objekt mit dem Wert 1. Die nächste Zeile, `y = 4`, bewirkt, dass `y` auf das Integer-Objekt `4` verweist, dies ändert jedoch nicht die Referenzen von `x` oder `z`.

Python-Variablen können auf jedes beliebige Objekt gesetzt werden, während in vielen anderen Sprachen Variablen nur im deklarierten Typ gespeichert werden können.

Variablennamen unterscheiden Groß- und Kleinschreibung und können jedes alphanumerische Zeichen sowie Unterstriche enthalten, müssen aber mit einem Buchstaben oder Unterstrich beginnen.

Bemerkung: Wenn ihr einen `SyntaxError` erhaltet, prüft, ob der Variablenname ein Schlüsselwort ist. Schlüsselwörter sind für die Verwendung in Python-Sprachkonstrukten reserviert, so dass ihr sie nicht zu Variablen machen könnt. Nach dem Aufruf von *help()* könnt ihr `keywords` eingeben, um die Schlüsselworte zu erhalten:

```
>>> 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
```

Bemerkung: Ihr könnt mit einem Variablennamen eingebaute (engl.: *built-in*) Funktionen, Typen und andere Objekte überschreiben, sodass der Zugriff anschließend nur noch über das builtins-Modul erfolgen kann. Daher sollten diese Variablennamen nie verwendet werden. Eine Liste der `__builtins__`-Objekte erhaltet ihr mit:

```
>>> 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__', '__
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪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 Ausdrücke

Python unterstützt arithmetische und ähnliche Ausdrücke. Der folgende Code berechnet den Durchschnitt von `x` und `y` und speichert das Ergebnis in der Variablen `z`:

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

Bemerkung: Arithmetische Operatoren, die nur ganze Zahlen verwenden, geben nicht immer eine ganze Zahl zurück. Ab Python 3 gibt die Division eine Fließkommazahl zurück. Wenn die traditionelle Ganzzahldivision mit einer Ganzzahl zurückgegeben werden soll, könnt ihr stattdessen `//` verwenden.

KAPITEL 7

Datentypen

Python verfügt über mehrere eingebaute Datentypen, wie z.B. *Zahlen* (Ganzzahlen, Gleitkommazahlen, komplexe Zahlen), *Zeichenketten*, *Listen*, *Tupel*, *Dictionaries*, *Sets* und *Dateien*. Diese Datentypen können mit Hilfe von Sprachoperatoren, eingebauten Funktionen, Bibliotheksfunktionen oder den eigenen Methoden eines Datentyps manipuliert werden.

Ihr könnt auch eure eigenen Klassen definieren und eigene Klasseninstanzen erstellen. Für diese Klasseninstanzen könnt ihr Methoden definieren sowie mit den Sprachoperatoren und eingebauten Funktionen, für die ihr die entsprechenden speziellen Methodenattribute definiert habt, bearbeitet werden.

Bemerkung: In der Python-Dokumentation und in diesem Buch wird der Begriff *Objekt* für Instanzen beliebiger Python-Datentypen verwendet, nicht nur für das, was viele andere Sprachen als Klasseninstanzen bezeichnen würden. Das liegt daran, dass alle Python-Objekte Instanzen der einen oder anderen Klasse sind.

Python hat mehrere eingebaute Datentypen, von Skalaren wie Zahlen und boolschen Werten bis hin zu komplexeren Strukturen wie Listen, Dictionaries und Dateien.

7.1 Zahlen

Die vier Zahlentypen von Python sind ganze Zahlen, Gleitkommazahlen, komplexe Zahlen und Boolesche Zahlen:

Typ	Beispiele
Ganzzahlen	`-1`, `42`, `90000000`
Gleitkommazahlen	`90000000.0`, `-0.005`, `9e7`, `-5e-3`
Komplexe Zahlen	`3 + 2j`, `-4- 2j`, `4.2 + 6.3j`
Boolesche Zahlen	`True`, `False`

Sie können mit den arithmetischen Operatoren manipuliert werden:

Operator	Beschreibung
+	Addition
-	Subtraktion
*	Multiplikation
/, //	Division[1]
**	Potenzierung
%	Modulus

Bemerkung: Ganzzahlen können unbegrenzt groß sein, begrenzt nur durch den verfügbaren Speicher.

Beispiele:

```
>>> 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
```

Siehe auch:

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

7.1.1 Komplexe Zahlen

Komplexe Zahlen bestehen aus einem Realteil und einem Imaginärteil, der in Python den Suffix `j` erhält.

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

Bemerkung: Python drückt die resultierende komplexe Zahl in Klammern aus, um anzuzeigen, dass die Ausgabe den Wert eines einzelnen Objekts darstellt:

```
>>> (7 + 2j) - (4 + 4j)
(3-2j)
```

```
>>> 2j * 4j
(-8+0j)
```

[1] Die Division ganzer Zahlen mit `/` führt zu einer Gleitkommazahl, und die Division ganzer Zahlen mit `//` führt zu einer Ganzzahl, die abgeschnitten wird.

Bemerkung: Die Berechnung von `2j * 4j` ergibt die erwartete Antwort von `-8`, aber das Ergebnis bleibt ein Python-Objekt für komplexe Zahlen. Komplexe Zahlen werden nie automatisch in entsprechende reelle oder ganzzahlige Objekte umgewandelt. Ihr könnt aber leicht auf ihre realen und imaginären Teile mit `real` und `imag` zugreifen.

```
>>> x = 2j * 4j
>>> x
(-8+0j)
>>> x.real
-8.0
>>> x.imag
0.0
```

Bemerkung: Die Real- und Imaginärteile einer komplexen Zahl werden immer als Fließkommazahlen zurückgegeben.

7.1.2 Built-in numerische Funktionen

Mehrere eingebaute Funktionen können mit Zahlen arbeiten:

abs()
: gibt den absoluten Wert einer Zahl zurück. Dabei kann das Argument eine Ganzzahl, eine Fließkommazahl oder ein Objekt sein, das `__abs__()` implementiert. Bei komplexen Zahlen als Argument wird ihr Betrag zurückgegeben.

divmod()
: nimmt zwei (nicht-komplexe) Zahlen als Argumente und gibt ein Zahlenpaar zurück, das aus ihrem Quotienten und dem Rest besteht, wenn eine ganzzahlige Division verwendet wird.

float
: Gibt eine Fließkommazahl zurück, die aus einer Zahl oder Zeichenkette `x` gebildet wird.

hex()
: konvertiert eine Integer-Zahl in eine klein geschriebene hexadezimale Zeichenkette mit dem Präfix `0x`.

int
: gibt ein Integer-Objekt zurück, das aus einer Zahl oder Zeichenkette `x` konstruiert wurde, oder `0`, wenn keine Argumente angegeben werden.

max()
: gibt das größte Element in einem iterable oder das größte von zwei oder mehr Argumenten zurück.

min()
: gibt das kleinste Element in einem Iterable oder das kleinste von zwei oder mehr Argumenten zurück.

oct()
: konvertiert eine Integer-Zahl in eine oktale Zeichenkette mit dem Präfix `0o`. Das Ergebnis ist ein gültiger Python-Ausdruck. Wenn `x` kein Python `int()`-Objekt ist, muss es eine `__index__()`-Methode definieren, die eine ganze Zahl zurückgibt.

pow()
: gibt *base* als Potenz von *exp* zurück.

round()
: gibt eine Zahl zurück, die auf *ndigits* nach dem Dezimalpunkt gerundet ist. Wird *ndigits* weggelassen oder ist *None*, wird die nächstgelegene Ganzzahl zur Eingabe zurückgegeben.

7.1.3 Boolsche Werte

In den folgenden Beispielen werden Boolesche Werte verwendet:

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

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

Abgesehen von ihrer Darstellung als `True` und `False` verhalten sich Boolesche Werte wie die Zahlen `1` (`True`) und `0` (`False`).

7.1.4 Erweiterte numerische Funktionen

Fortgeschrittenere numerische Funktionen wie Trigonometrie sowie einige nützliche Konstanten sind nicht in Python integriert, sondern werden in einem Standardmodul namens math bereitgestellt. *Module* werden später noch ausführlicher erklärt. Für den Moment genügt, dass ihr die mathematischen Funktionen in diesem Abschnitt verfügbar machen müsst, indem ihr `math` importiert:

```
import math
```

Eingebaute Funktionen sind immer verfügbar und werden mit einer Standard-Syntax für Funktionsaufrufe aufgerufen. Im folgenden Code wird `round` mit einem Float als Eingangsargument aufgerufen.

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

Mit `ceil` aus der Standardbibliothek `math` und der Attributschreibweise `MODUL.FUNKTION(ARGUMENT)` wird aufgerundet:

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

Das `math`-Modul bietet U.A.

- die zahlentheoretischen und Darstellungsfunktionen `math.ceil()`, `math.modf()`, `math.frexp()` und `math.ldexp()`,
- die Potenz- und logarithmische Funktionen `math.exp()`, `math.log()`, `math.log10()`, `math.pow()` und `math.sqrt()`,
- die trigonometrischen Funktionen `math.acos()`, `math.asin()`, `math.atan()`, `math.atan2()`, `math.ceil()`, `math.cos()`, `math.hypot()` und `math.sin()`,
- die hyperbolischen Funktionen `math.cosh()`, `math.sinh()` und `math.tanh()`
- und die Konstanten `math.e` und `math.pi`.

7.1.5 Erweiterte Funktionen für komplexe Zahlen

Die Funktionen im Modul math sind nicht auf komplexe Zahlen anwendbar; einer der Gründe hierfür dürfte sein, dass die Quadratwurzel aus -1 einen Fehler erzeugen soll. Daher wurden ähnliche Funktionen für komplexe Zahlen arbeiten im cmath-Modul bereitgestellt:

`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()`.

Um im Code deutlich zu machen, dass es sich bei diesen Funktionen um spezielle Funktionen für komplexe Zahlen handelt, und um Namenskonflikte mit den normaleren Äquivalenten zu vermeiden, empfiehlt sich der einfache Import des Moduls um bei der Verwendung der Funktion ausdrücklich auf das `cmath`-Paket zu verweisen, z.B.:

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

Warnung: Nun wird auch verständlicher, weswegen wir nicht den Import aller Funktionen eines Moduls empfehlen mit `from MODULE import *`. Wenn ihr damit zuerst das Modul `math` und dann das Modul `cmath` importieren würdet, hätten die Funktionen in `cmath` Vorrang vor denen von `math`. Außerdem ist es beim Verstehen des Codes viel mühsamer, die Quelle der verwendeten Funktionen herauszufinden.

7.1.6 Kaufmännisch runden

Üblicherweise rechnet Python Gleitkommazahlen der IEEE 754-Norm entsprechend, wobei Zahlen in der Mitte in der Hälfte der Fälle abgerundet werden und in der anderen Hälfte aufgerundet werden um eine statistische Drift bei längeren Rechnungen zu vermeiden. Für das kaufmännische Runden werden daher `Decimal` und `ROUND_HALF_UP` aus dem `decimal`-Modul benötigt:

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

7.1.7 Numerische Berechnungen

Die Python-Standardinstallation eignet sich aufgrund von Geschwindigkeitseinschränkungen nicht gut für intensive numerische Berechnungen. Aber die leistungsstarke Python-Erweiterung NumPy bieten hocheffiziente Implementierungen vieler fortgeschrittener numerischer Operationen. Der Schwerpunkt liegt dabei auf Array-Operationen, einschließlich mehrdimensionaler Matrizen und fortgeschrittener Funktionen wie der schnellen Fourier-Transformation.

7.1.8 Eingebaute Module für Zahlen

Die Python-Standardbibliothek enthält eine Reihe eingebauter Module, mit denen ihr Zahlen managen könnt:

Modul	Beschreibung
`numbers`	für numerische abstrakte Basisklassen
`math`, `cmath`	für mathematische Funktionen für reelle und komplexe Zahlen
`decimal`	für dezimale Festkomma- und Gleitkomma-Arithmetik
`statistics`	für Funktionen zur Berechnung von mathematischen Statistiken
`fractions`	für rationale Zahlen
`random`	zum Erzeugen von Pseudozufallszahlen und -auswahlen sowie zum Mischen von Sequenzen
`itertools`	für Funktionen, die Iteratoren für effiziente Schleifen erzeugen
`functools`	für Funktionen höherer Ordnung und Operationen auf aufrufbaren Objekten
`operator`	für Standardoperatoren als Funktionen

7.2 Listen

Python hat einen mächtigen eingebauten Listentyp:

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

Eine Liste kann eine Mischung anderer Typen als Elemente enthalten, darunter Zeichenketten, Tupel, Listen, Dictionaries, Funktionen, Dateiobjekte und jede Art von Zahl.

Eine Liste kann von vorne oder hinten indiziert werden. Ihr könnt euch auch auf ein Teilsegment einer Liste beziehen, indem ihr die Slice-Notation verwendet:

```
1  >>> x = [1, "2.", 3.0, ["4a", "4b"], (5.1, 5.2)]
2  >>> x[0]
3  '1'
4  >>> x[1]
5  '2.'
6  >>> x[-1]
7  (5.1, 5.2)
8  >>> x[-2]
9  ['4a', '4b']
10 >>> x[1:-1]
11 ['2.', 3.0, ['4a', '4b']]
12 >>> x[0:3]
13 [1, '2.', 3.0]
14 >>> x[:3]
15 [1, '2.', 3.0]
16 >>> x[-4:-1]
17 ['2.', 3.0, ['4a', '4b']]
18 >>> x[-4:]
19 ['2.', 3.0, ['4a', '4b'], (5.1, 5.2)]
```

Zeilen 2 und 4
Index von vorne unter Verwendung positiver Indizes beginnend mit `0` als erstem Element.

Zeilen 6 und 8
Index von hinten unter Verwendung negativer Indizes beginnend mit `-1` als letztem Element.

Zeilen 10 und 12
Slice mit `[m:n]`, wobei `m` der inklusive Startpunkt und `n` der exklusive Endpunkt ist.

Zeilen 14, 16 und 18
Ein `[:n]`-Slice beginnt am Anfang und ein `[m:]`-Slice geht bis zum Ende einer Liste.

Ihr könnt diese Notation verwenden, um Elemente in einer Liste hinzuzufügen, zu entfernen und zu ersetzen oder um ein Element oder eine neue Liste zu erhalten, die ein Slice davon ist, z.B.:

```
>>> 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)]
```

Zeile 3
Die Größe der Liste erhöht oder verringert sich, wenn das neue Slice größer oder kleiner ist als das Slice, das es ersetzt.

Slices erlauben auch eine stufenweise Auswahl zwischen den Start- und Endindizes. Der Standardwert für ein nicht spezifiziertes Stride ist 1, womit jedes Element aus einer Sequenz zwischen den Indizes genommen wird. Bei einem Stride von 2 wird jedes zweite Element übernommen usw.:

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

Der Stride-Wert kann auch negativ sein. Ein `-1`-Stride bedeutet, von rechts nach links gezählt wird:

```
>>> 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]
```

Zeile 1
Um eine negative Schrittweite zu verwenden, sollte das Start-Slice größer sein als das End-Slice.

Zeile 3
Die Ausnahme ist, wenn ihr die Start- und Endindizes weglasst.

Zeile 5
Ein Stride von `-1` kehrt die Reihenfolge um.

Einige Funktionen der Slice-Notation können auch mit speziellen Operationen wausgeführt werden, wodurch die Lesbarkeit des Codes verbessert wird:

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

Darüberhinaus könnt ihr die fogenden eingebauten Funktionen (`len`, `max` und `min`), einige Operatoren (`in`, + und *), die `del`-Anweisung und die Listenmethoden (`append`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse` und `sort`) für Listen verwenden:

```
>>> 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)]
```

Zeile 3
Die Operatoren + und * erzeugen jeweils eine neue Liste, wobei die ursprüngliche Liste unverändert bleibt.

Zeile 5
Die Methoden einer Liste werden mit Hilfe der Attributschreibweise für die Liste selbst aufgerufen: *`LISTE`*`.`*`METHODE`*`(`*`ARGUMENTE`*`)`.

Siehe auch:

- Daten auswählen und filtern mit pandas

7.2.1 Zusammenfassung

Datentyp	veränderlich	geordnet	indiziert	Duplikate
Liste				

7.3 Tupel

Tupel ähneln Listen, sind aber unveränderlich, d.h. sie können nach ihrer Erstellung nicht mehr geändert werden. Die Operatoren (`in`, + und *) und eingebauten Funktionen (`len`, `max` und `min`) arbeiten mit ihnen auf die gleiche Weise wie mit *Listen*, da keine dieser Funktionen das Original verändert. Die Index- und die Slice-Notation funktionieren auf die gleiche Weise, um Elemente oder Slices zu erhalten, können aber nicht zum Hinzufügen, Entfernen oder Ersetzen von Elementen verwendet werden. Außerdem gibt es nur zwei Tupelmethoden: `count` und `index`. Ein wichtiger Zweck von Tupeln ist die Verwendung als Schlüssel für *Dictionaries*. Sie sind auch effizienter zu verwenden, wenn man keine Änderungsmöglichkeit benötigt.

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

Zeile 2
Ein Tupel mit einem Element benötigt ein Komma.

Zeile 4
Ein Tupel kann, wie eine *Liste*, eine Mischung anderer Typen als Elemente enthalten, darunter beliebige *Zahlen*, *Zeichenketten*, *Tupel*, *Listen*, *Dictionaries*, *Dateien* und Funktionen.

Eine Liste kann mit Hilfe der eingebauten Funktion `tuple` in ein Tupel umgewandelt werden:

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

Umgekehrt kann ein Tupel mit Hilfe der eingebauten Funktion list in eine Liste umgewandelt werden:

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

Die Vorteile von Tupeln gegenüber *Listen* sind:

- Tupel sind schneller als Listen.

  Wenn ihr eine konstante Menge von Werten definieren und diese nur durchlaufen wollt, solltet ihr ein Tupel anstelle einer Liste verwenden.

- Tupel können nicht verändert werden und sind daher *schreibgeschützt*.
- Tupel können als Schlüssel in *Dictionaries* und Werte in *Sets* verwendet werden.

7.3.1 Zusammenfassung

Datentyp	veränderlich	geordnet	indiziert	Duplikate
Tuple				

7.4 Sets

Ein Set in Python ist eine ungeordnete Sammlung von Objekten, die in Situationen verwendet wird, in denen die Zugehörigkeit und Einzigartigkeit zur Menge die wichtigsten Informationen des Objekts sind. Der `in`-Operator läuft bei Sets schneller als bei *Listen*:

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

Zeile 1
Ihr könnt ein Set erstellen, indem ihr `set` auf eine Sequenz wie eine *Liste* anwendet.

Zeile 3
Wenn eine Sequenz zu einem Set gemacht wird, werden Duplikate entfernt.

Zeilen 4 und 6
Das Schlüsselwort wird verwendet, um die Zugehörigkeit eines Objekts zu einer Menge zu prüfen.

Sets verhalten sich wie Kollektionen von *Dictionary*-Schlüsseln ohne zugehörige Werte.

Der Geschwindigkeitsvorteil hat jedoch auch ihren Preis: Sets halten die Elemente nicht in der richtigen Reihenfolge, während *Listen* und *Tupel* dies tun. Wenn die Reihenfolge für euch wichtig ist, solltet ihr eine Datenstruktur verwenden, die sich die Reihenfolge merkt.

7.4.1 Zusammenfassung

Datentyp	veränderlich	geordnet	indiziert	Duplikate
Set				

7.5 Dictionaries

Pythons eingebauter Dictionary-Datentyp bietet assoziative Array-Funktionalität, die mit Hilfe von Hash-Tabellen implementiert wird. Die eingebaute Funktion `len` gibt die Anzahl der Schlüssel-Wert-Paare in einem Wörterbuch zurück. Die `del`-Anweisung kann zum Löschen eines Schlüssel-Wert-Paares verwendet werden. Wie bei *Listen* sind mehrere Dictionary-Methoden (`clear`, `copy`, `get`, `items`, `keys`, `update` und `values`) verfügbar.

```
>>> x = {1: "eins", 2: "zwei"}
>>> x[3] = "drei"
>>> x["viertes"] = "vier"
>>> list(x.keys())
[1, 2, 3, 'viertes']
>>> x[1]
'eins'
>>> x.get(1, "nicht vorhanden")
'eins'
>>> x.get(5, "nicht vorhanden")
'nicht vorhanden'
```

Schlüssel müssen vom unveränderlichen Typ sein, einschließlich *Zahlen*, *Zeichenketten* und Tupel.

Warnung: Auch wenn ihr in einem Dictionary verschiedene Schlüsseltypen verwenden könnt, solltet ihr das vermeiden, da dadurch nicht nur die Lesbarkeit sondern auch die Sortierung erschwert wird.

Werte können alle Arten von Objekten sein, einschließlich veränderlicher Typen wie *Listen* und *Dictionaries*. Wenn ihr versucht, auf den Wert eines Schlüssels zuzugreifen, der nicht im Dictionary enthalten ist, wird eine `KeyError`-Exception ausgelöst. Um diesen Fehler zu vermeiden, gibt die Dictionary-Methode `get` optional einen benutzerdefinierten Wert zurück, wenn ein Schlüssel nicht in einem Wörterbuch enthalten ist.

7.5.1 setdefault

`setdefault` kann verwendet werden, um Zähler für die Schlüssel eines Dicts bereitzustellen, z.B.:

```
>>> 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}
```

Bemerkung: Solche Zähloperationen verbreiteten sich schnell, sodass später die Klasse `collections.Counter` zur Python-Standardbibliothek hinzugefügt wurde. Diese Klasse kann die oben genannten Operationen viel einfacher durchführen:

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

7.5.2 Dictionaries zusammenführen

Ihr könnt zwei Dictionaries zu einem einzigen Dictionary zusammenfügen mit der `dict.update()`-Methode:

```
>>> 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'}
```

Bemerkung: Die Reihenfolge der Operanden ist wichtig, da `7.0` dupliziert wird und der Wert des letzten Schlüssel den vorhergehenden überschreibt.

7.5.3 Erweiterungen

python-benedict
: `dict`-Unterklasse mit Keylist/Keypath/Keyattr-Unterstützung sowie I/O-Shortcuts.

pandas
: kann Dicts in Series und DataFrames überführen.

7.6 Zeichenketten

Die Verarbeitung von Zeichenketten ist eine der Stärken von Python. Es gibt viele Optionen zur Begrenzung von Zeichenketten:

```
"Eine Zeichenfolge in doppelten Anführungszeichen kann 'einfache Anführungszeichen'␣
↪enthalten."
'Eine Zeichenfolge in einfachen Anführungszeichen kann "doppelte Anführungszeichen"␣
↪enthalten.'
"""\tEine Zeichenkette, die mit einem Tabulator beginnt und mit einem␣
↪Zeilenumbruchzeichen endet.\n"""
"""Dies ist eine Zeichenkette in dreifach doppelten Anführungszeichen, die
einzige Zeichenkette, die echte Zeilenumbrüche enthält."""
```

Zeichenketten können durch einfache (`' '`), doppelte (`" "`), dreifache einfache (`''' '''`) oder dreifache doppelte (`""" """`) Anführungszeichen getrennt werden und können Tabulator- (`\t`) und Zeilenumbruchzeichen (`\n`) enthalten. Allgemein können Backslashes \ als Escape-Zeichen verwendet werden. So kann z.B. `\\` für einen einzelnen Backslash und `\'` für ein einfaches Anführungszeichen verwendet werden, wodurch es die Zeichenfolge nicht beendet:

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

Hier sind weitere Zeichen, die ihr mit dem Escape-Zeichen erhalten könnt:

Escape-Sequenz	Ausgabe	Erläuterung
`\\`	\	Backslash
`\'`	'	einfaches Anführungszeichen
`\"`	"	doppeltes Anführungszeichen
`\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

Eine normale Zeichenkette kann nicht auf mehrere Zeilen aufgeteilt werden. Der folgende Code wird nicht funktionieren:

```
"Dies ist ein fehlerhafter Versuch, einen einen Zeilenumbruch in
eine Zeichenkette einzufügen, ohne \n zu verwenden."
```

Python bietet jedoch Zeichenketten in dreifachen Anführungszeichen (`"""`), die dies ermöglichen und einfache und doppelte Anführungszeichen ohne Backslashes enthalten können.

Zeichenketten sind außerdem unveränderlich. Die Operatoren und Funktionen, die mit ihnen arbeiten, geben neue, vom Original abgeleitete Zeichenketten zurück. Die Operatoren (`in`, `+` und `*`) und eingebauten Funktionen (`len`, `max` und `min`) arbeiten mit Zeichenketten genauso wie mit Listen und Tupeln.

```
>>> welcome = "Hello pythonistas!\n"
>>> 2 * welcome
'Hello pythonistas!\nHello pythonistas!\n'
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

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

Die Index- und Slice-Notation funktioniert auf die gleiche Weise, um Elemente oder Slices zu erhalten:

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

Die Index- und Slice-Notation kann jedoch nicht verwendet werden, um Elemente hinzuzufügen, zu entfernen oder zu ersetzen:

```
>>> 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

Für Zeichenketten gibt es in der Standard-Python-Bibliothek string mehrere Methoden, um mit ihrem Inhalt zu arbeiten u.a. `str.split()`, `str.replace()` und `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!']
>>> " ".join(chunks)
'hello pythonistas!'
>>> welcome.replace("\n", "")
'hello pythonistas!'
```

Im Folgenden findet ihr einen Überblick über die häufigsten String-Methoden:

Methode	Beschreibung
`str.count()`	gibt die Anzahl der sich nicht überschneidenden Vorkommen der Zeichenkette zurück.
`str.endswith()`	gibt `True` zurück, wenn die Zeichenkette mit dem Suffix endet.
`str.startswith()`	gibt `True` zurück, wenn die Zeichenkette mit dem Präfix beginnt.
`str.join()`	verwendet die Zeichenkette als Begrenzer für die Verkettung einer Folge anderer Zeichenketten.
`str.index()`	gibt die Position des ersten Zeichens in der Zeichenkette zurück, wenn es in der Zeichenkette gefunden wurde; löst einen `ValueError` aus, wenn es nicht gefunden wurde.
`str.find()`	gibt die Position des ersten Zeichens des ersten Vorkommens der Teilzeichenkette in der Zeichenkette zurück; wie `index`, gibt aber `-1` zurück, wenn nichts gefunden wurde.
`str.rfind()`	Rückgabe der Position des ersten Zeichens des letzten Vorkommens der Teilzeichenkette in der Zeichenkette; gibt `-1` zurück, wenn nichts gefunden wurde.
`str.replace()`	ersetzt Vorkommen einer Zeichenkette durch eine andere Zeichenkette.
`str.strip()`, `str.rstrip()`, `str.lstrip()`	schneiden Leerzeichen ab, einschließlich Zeilenumbrüchen.
`str.split()`	zerlegt eine Zeichenkette in eine Liste von Teilzeichenketten unter Verwendung des übergebenen Trennzeichens.
`str.lower()`	konvertiert alphabetische Zeichen in Kleinbuchstaben.
`str.upper()`	konvertiert alphabetische Zeichen in Großbuchstaben.
`str.casefold()`	konvertiert Zeichen in Kleinbuchstaben und konvertiert alle regionsspezifischen variablen Zeichenkombinationen in eine gemeinsame vergleichbare Form.
`str.ljust()`, `str.rjust()`	linksbündig bzw. rechtsbündig; füllt die gegenüberliegende Seite der Zeichenkette mit Leerzeichen (oder einem anderen Füllzeichen) auf, um eine Zeichenkette mit einer Mindestbreite zu erhalten.

Darüber hinaus gibt es einige Methoden, mit denen die Eigenschaft einer Zeichenkette überprüft werden kann:

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

`str.isspace()` prüft auf Leerzeichen: `[ \t\n\r\f\v\x1c-\x1f\x85\xa0\u1680...]`.

7.6.2 `re`

Die Python-Standard-Bibliothek re enthält ebenfalls Funktionen für die Arbeit mit Zeichenketten. Dabei bietet `re` ausgefeiltere Möglichkeiten zur Musterextraktion und -ersetzung als `string`.

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

Hier wird der reguläre Ausdruck zunächst kompiliert und dann seine `re.Pattern.sub()`-Methode für den übergebenen Text aufgerufen. Ihr könnt den Ausdruck selbst mit `re.compile()` kompilieren und so ein wiederverwendbares `regex`-Objekt bilden, das auf unterschiedliche Zeichenketten angewendet die CPU-Zyklen verringert:

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

Wenn ihr stattdessen eine Liste aller Muster erhalten möchtet, die dem `regex`-Objekt entsprechen, könnt ihr die `re.Pattern.findall()`-Methode verwenden:

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

Bemerkung: Um das umständliche Escaping mit \ in einem regulären Ausdruck zu vermeiden, könnt ihr rohe String-Literale wie `r'C:\PATH\TO\FILE'` anstelle des entsprechenden `'C:\\PATH\\TO\\FILE'` verwenden.

`re.Pattern.match()` und `re.Pattern.search()` sind eng mit `re.Pattern.findall()` verwandt. Während `findall` alle Übereinstimmungen in einer Zeichenkette zurückgibt, gibt `search` nur die erste Übereinstimmung und `match` nur Übereinstimmungen am Anfang der Zeichenkette zurück. Als weniger triviales Beispiel betrachten wir einen Textblock und einen regulären Ausdruck, der die meisten E-Mail-Adressen identifizieren kann:

```
>>> 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` gibt `None` zurück, da das Muster nur dann passt, wenn es am Anfang der Zeichenkette steht.

Angenommen, ihr möchtet E-Mail-Adressen finden und gleichzeitig jede Adresse in ihre drei Komponenten aufteilen:

1. Personenname
2. Domänenname
3. Domänensuffix

Dazu setzt ihr zunächst runde Klammern `()` um die zu segmentierenden Teile des Musters:

```
>>> 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()` gibt ein *Tupel* zurück, das alle Untergruppen der Übereinstimmung enthält.

`re.Pattern.findall()` gibt eine Liste von Tupeln zurück, wenn das Muster Gruppen enthält:

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

Auch in `re.Pattern.sub()` können Gruppen verwendet werden wobei `\1` für die erste übereinstimmende Gruppe steht, `\2` für die zweite usw.:

```
>>> 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>
```

Die folgende Tabelle enthält einen kurzen Überblick über Methoden für reguläre Ausdrücke:

Methode	Beschreibung
`re.findall()`	gibt alle sich nicht überschneidenden übereinstimmenden Muster in einer Zeichenkette als Liste zurück.
`re.finditer()`	wie `findall`, gibt aber einen Iterator zurück.
`re.match()`	entspricht dem Muster am Anfang der Zeichenkette und segmentiert optional die Musterkomponenten in Gruppen; wenn das Muster übereinstimmt, wird ein `match`-Objekt zurückgegeben, andernfalls keines.
`re.search()`	durchsucht die Zeichenkette nach Übereinstimmungen mit dem Muster; gibt in diesem Fall ein `match`-Objekt zurück; im Gegensatz zu `match` kann die Übereinstimmung an einer beliebigen Stelle der Zeichenkette und nicht nur am Anfang stehen.
`re.split()`	zerlegt die Zeichenkette bei jedem Auftreten des Musters in Teile.
`re.sub()`, `re.subn()`	ersetzt alle (`sub`) oder die ersten `n` Vorkommen (`subn`) des Musters in der Zeichenkette durch einen Ersetzungsausdruck; verwendet die Symbole `\1`, `\2`, ..., um auf die Elemente der Übereinstimmungsgruppe zu verweisen.
`str.removeprefix()` `str.removesuffix()`	In Python 3.9 kann dies verwendet werden, um das Suffix oder den Dateinamen zu extrahieren.

Siehe auch:

- *Reguläre Ausdrücke*
- Regular Expression HOWTO
- re — Regular expression operations

7.6.3 `print()`

Die Funktion `print()` gibt Zeichenketten aus wobei andere Python-Datentypen leicht in Strings umgewandelt und formatiert werden können, z.B.:

```
>>> import math
>>> pi = math.pi
>>> d = 28
>>> u = pi * d
>>> print(
...     "Pi ist",
...     pi,
...     "und der Umfang bei einem Durchmesser von",
...     d,
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
...     "Zoll ist",
...     u,
...     "Zoll.",
... )
Pi ist 3.141592653589793 und der Umfang bei einem Durchmesser von 28 Zoll ist 87.
↪96459430051421 Zoll.
```

F-Strings

Mit F-Strings lassen sich die für einen Text zu detaillierten Zahlen kürzen:

```
>>> print(f"Der Wert von Pi ist {pi:.3f}.")
Der Wert von Pi ist 3.142.
```

In `{pi:.3f}` wird die Format-Spezifikation `f` verwendet, um die Zahl Pi auf drei Nachkommastellen zu kürzen.

In A/B-Testszenarien möchtet ihr oft die prozentuale Veränderung einer Kennzahl darstellen. Mit F-Strings können sie verständlich formuliert werden:

```
>>> metrics = 0.814172
>>> print(f"Die AUC hat sich vergrößert auf {metrics:=+7.2%}")
Die AUC hat sich vergrößert auf +81.42%
```

In diesem Beispiel wird die Variable `metrics` formatiert, wobei = die Inhalte der Variable nach dem + übernimmt, wobei insgesamt sieben Zeichen einschließlich des Vorzeichen, `metrics` und des Prozentzeichens angezeigt werden. `.2` sorgt für zwei Dezimalstellen, während das `%`-Symbol den Dezimalwert in eine Prozentzahl umwandelt. So wird `0.514172` in `+51.42%` umgewandelt.

Werte lassen sich auch in binäre und hexadezimale Werte umrechnen:

```
>>> 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
```

Es gibt auch Formatierungsangaben, die ideal geeignet sind für die CLI (Command Line Interface)-Ausgabe, z.B.:

```
>>> 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
```

Allgemein sieht das Format folgendermaßen aus, wobei alle Angaben in eckigen Klammern optional sind:

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

In der folgenden Tabelle sind die Felder für die Zeichenkettenformatierung und ihre Bedeutung aufgeführt:

Feld	Bedeutung
FILL	Zeichen, das zum Ausfüllen von ALIGN verwendet wird. Der Standardwert ist ein Leerzeichen.
ALIGN	Textausrichtung und Füllzeichen: <: linksbündig >: rechtsbündig ^: zentriert =: Füllzeichen nach SIGN
SIGN	Vorzeichen anzeigen: +: Vorzeichen bei positiven und negativen Zahlen anzeigen -: Standardwert, - nur bei negativen Zahlen oder Leerzeichen bei positiven Zahlen
0b|0o|0x|d|n	Vorzeichen für ganze Zahlen: 0b: Binärzahlen 0o: Oktalzahlen 0x: Hexadezimalzahlen d: Standardwert, dezimale Ganzzahl zur Basis 10 n: verwendet die aktuelle locale-Einstellung, um die entsprechenden Zahlentrennzeichen einzufügen
0	füllt mit Nullen auf
WIDTH	Minimale Feldbreite
GROUPING	Zahlentrennzeichen:[1] ,: Komma als Tausendertrennzeichen _: Unterstrich für Tausendertrennzeichen
.PRECISION	 Bei Fließkommazahlen die Anzahl der Ziffern nach dem Punkt bei nicht-numerischen Werten die maximale Länge
TYPE	Ausgabeformat als Zahlentyp oder Zeichenkette ... für Ganzzahlen: b: Binärformat c: konvertiert die Ganzzahl in das entsprechende Unicode-Zeichen d: Standardwert, Dezimalzeichen n: dasselbe wie d, mit dem Unterschied, dass es die aktuelle locale-Einstellung verwendet, um die entsprechenden Zahlentrennzeichen einzufügen o: Oktalformat x: Hexadezimalformat zur Basis 16, wobei für die Ziffern über 9 Kleinbuchstaben verwendet werden X: Hexadezimalformat zur Basis 16, wobei für die Ziffern über 9 Großbuchstaben verwendet werden ... für Fließkommazahlen:

Tipp: Eine gute Quelle für F-Strings ist die Hilfe-Funktion:

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

Ihr könnt die Hilfe hier durchblättern und viele Beispiele finden.

Mit `:–q` und könnt ihr die Hilfe-Funktion wieder verlassen.

Siehe auch:

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

Fehlersuche in F-Strings

In Python 3.8 wurde ein Spezifizierer eingeführt, der bei der Fehlersuche in F-String-Variablen hilft. Durch Hinzufügen eines Gleichheitszeichens = wird der Code innerhalb des F-Strings aufgenommen:

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

Formatierung von Datums-, Zeitformaten und IP-Adressen

`datetime` unterstützt die Formatierung von Zeichenketten mit der gleichen Syntax wie die `strftime`-Methode für diese Objekte.

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

Das `ipaddress`-Modul von Python unterstützt auch die Formatierung von `IPv4Address`- und `IPv6Address`-Objekten.

Schließlich können Bibliotheken von Drittanbietern auch ihre eigene Unterstützung für die Formatierung von Strings hinzufügen, indem sie eine `__format__`-Methode zu ihren Objekten hinzufügen.

Siehe auch:

- strftime() and strptime() Format Codes

[1] Der Formatbezeichner `n` formatiert eine Zahl in einer lokal angepassten Weise, z.B.:

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

- Python strftime cheatsheet

7.6.4 Eingebaute Module für Zeichenketten

Die Python-Standardbibliothek enthält eine Reihe eingebauter Module, mit denen ihr Zeichenketten managen könnt:

Modul	Beschreibung
`string`	vergleicht mit Konstanten wie `string.digits` oder `string.whitespace`
`re`	sucht und ersetzt Text mit regulären Ausdrücken
`struct`	interpretiert Bytes als gepackte Binärdaten
`difflib`	hilft beim Berechnen von Deltas, beim Auffinden von Unterschieden zwischen Zeichenketten oder Sequenzen und beim Erstellen von Patches und Diff-Dateien
`textwra`	umbricht und füllt Text, formatiert Text mit Zeilenumbrüchen oder Leerzeichen

Siehe auch:

- Manipulation von Zeichenketten mit pandas

7.7 Dateien

7.7.1 Öffnen von Dateien

In Python öffnet und lest ihr eine Datei, indem ihr die eingebaute Funktion `open()` und verschiedene eingebaute Leseoperationen verwendet. Das folgende kurze Python-Programm liest eine Zeile aus einer Textdatei namens *`myfile.txt`* ein:

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

`open()` liest nichts aus der Datei, sondern gibt ein SOG. (sogenanntes) Datei-Objekt zurück, mit dem ihr auf die geöffnete Datei zugreifen könnt. Es behält den Überblick über eine Datei und darüber, wie viel von der Datei gelesen oder geschrieben wurde. Alle Dateieingaben in Python werden mit Dateiobjekten und nicht mit Dateinamen durchgeführt.

Der erste Aufruf von `readline` gibt die erste Zeile des Datei-Objekts zurück, also alles bis einschließlich des ersten Zeilenumbruchs oder die gesamte Datei, wenn es keinen Zeilenumbruch in der Datei gibt; der nächste Aufruf von `readline` gibt die zweite Zeile zurück, wenn sie existiert, USW (und so weiter).

Das erste Argument der Funktion `open` ist ein Pfadname. Im vorigen Beispiel öffnet ihr eine Datei, von der ihr annehmt, dass sie sich im aktuellen Arbeitsverzeichnis befindet. Das folgende Beispiel öffnet eine Datei an einem absoluten Speicherort – *`C:Meine Dokumente\myfile.txt`*:

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

Bemerkung: In diesem Beispiel wird das Schlüsselwort `with` verwendet, D.H., dass die Datei mit einem Kontextmanager geöffnet wird, der in *Kontextmanagement mit with* näher erläutert wird. Diese Art des Öffnens von Dateien verwaltet mögliche I/O-Fehler besser und sollte im Allgemeinen bevorzugt werden.

7.7.2 Schließen von Dateien

Nachdem alle Daten aus einem Datei-Objekt gelesen oder in dieses geschrieben wurden, sollte das Datei-Pbjekt wieder geschlossen werden damit Systemressourcen freigegeben werden, das Lesen oder Schreiben der zugrunde liegenden Datei durch anderen Code ermöglicht wird und das Programm insgesamt zuverlässiger wird. Bei kleinen Skripten hat dies in der Regel keine großen Auswirkungen, da Dateiobjekte werden automatisch geschlossen, wenn das Skript oder Programm beendet wird. Bei größeren Programmen können zu viele offene Datei-Objekte jedoch die Systemressourcen erschöpfen, was zum Abbruch des Programms führen. Ihr schließt ein Dateiobjekt mit der `close`-Methode, wenn das Datei-Objekt nicht mehr benötigt wird:

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

Die Verwendung eines *Kontextmanagement mit with* bleibt meist jedoch die bessere Möglichkeit, um Dateien automatisch zu schließen, wenn ihr fertig seid:

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

7.7.3 Öffnen von Dateien im Schreib- oder anderen Modi

Das zweite Argument des Befehls `open()` ist eine Zeichenkette, die angibt, wie die Datei geöffnet werden soll. `"r"` öffnet die Datei zum Lesen (engl. *read*), `"w"` öffnet die Datei zum Schreiben (engl. *write*) und `"a"` offnet die Datei zum Anhängen (engl. *attach*). Wenn ihr die Datei zum Lesen öffnen wollen, könnt ihr das zweite Argument weglassen, da `"r"` der Standardwert ist. Das folgende kurze Programm schreibt `Hi, Pythonistas!` in eine Datei:

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

Je nach Betriebssystem kann `open()` auch Zugang zu weiteren Dateimodi haben. Diese Modi sind jedoch für die meisten Zwecke nicht notwendig.

`open` kann ein optionales drittes Argument annehmen, das definiert, wie Lese- oder Schreibvorgänge für diese Datei gepuffert werden. Beim Puffern werden Daten so lange im Speicher gehalten, bis genügend Daten angefordert oder geschrieben wurden, um die Zeitaufwände für einen Plattenzugriff zu rechtfertigen. Andere Parameter für `open` steuern die Kodierung für Textdateien und die Behandlung von Zeilenumbrüchen in Textdateien. Auch hier gilt, dass ihr euch in der Regel keine Gedanken über diese Funktionen machen müsst, aber wenn ihr mit Python fortgeschrittener werdet, solltet ihr euch vielleicht darüber informieren.

7.7.4 Lese- und Schreib-Funktionen

Die häufigste Funktion zum Lesen von Textdateien, `readline`, habe ich bereits vorgestellt. Diese Funktion liest eine einzelne Zeile aus einem Datei-Objekt und gibt sie zurück, einschließlich aller Zeilenumbrüche am Ende der Zeile. Wenn es nichts mehr zu lesen gibt, gibt readline einen leeren String zurück, was es einfach macht, z.B. die Anzahl der Zeilen in einer Datei zu ermitteln:

```
>>> f = open("docs/types/myfile.txt", "r")
>>> lc = 0
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
>>> while f.readline() != "":
...     lc = lc + 1
...
>>> print(lc)
2
>>> f.close()
```

Ein kürzerer Weg, alle Zeilen zu zählen, gibt es mit der ebenfalls eingebauten `readlines`-Methode, die alle Zeilen einer Datei liest und sie als Liste von Strings mit einen String pro Zeile zurückgibt:

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

Wenn ihr alle Zeilen einer großen Datei zählt, kann diese Methode dazu führen, dass der Speicher vollläuft, weil die gesamte Datei auf einmal geliesen wird. Es ist auch möglich, dass der Speicher mit `readline` überläuft, wenn ihr versucht, eine Zeile aus einer großen Datei zu lesen, die keine Zeilenumbruchzeichen enthältist. Um mit solchen Situationen besser umgehen zu können, haben beide Methoden ein optionales Argument, das die Menge der zu einem Zeitpunkt gelesenen Daten beeinflusst. Eine andere Möglichkeit, über alle Zeilen einer Datei zu iterieren, besteht darin, das Dateiobjekt als Iterator in einer *for-Schleife* zu behandeln:

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

Diese Methode hat den Vorteil, dass die Zeilen je nach Bedarf in den Speicher eingelesen werden, so dass selbst bei großen Dateien kein Speicherplatzmangel zu befürchten ist. Der andere Vorteil dieser Methode ist, dass sie einfacher und lesbarer ist.

Ein mögliches Problem mit der Lesemethode kann jedoch entstehen, wenn auf Windows- und macOS Übersetzungen im Textmodus erfolgen, wenn ihr den Befehl `open()` im Textmodus verwenden, D.H. ohne ein `b` anzuhängen. Im Textmodus wird auf macOS jedes `\r` in `\n` umgewandelt, während unter Windows `\r\n`-Paare in `\n` umgewandelt werden. Ihr könnt die Behandlung von Zeilenumbrüchen festlegen, indem ihr beim Öffnen der Datei den Parameter `newline` verwendet und `newline="\n"`, `\r` oder `\r\n` angebt, wodurch nur diese Zeichenfolge als Zeilenumbruch verwendet wird:

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

In diesem Beispiel wird nur `\n` als Zeilenumbruch gewertet. Wenn die Datei jedoch im Binärmodus geöffnet wurde, ist der Parameter `newline` nicht erforderlich, da alle Bytes genau so zurückgegeben werden, wie sie in der Datei stehen.

Die Schreibmethoden, die den Methoden `readline` und `readlines` entsprechen, sind `write` und `writelines`. Beachtet, dass es keine `writeline`-Funktion gibt. `write` schreibt eine einzelne Zeichenkette, die sich über mehrere Zeilen erstrecken kann, wenn Zeilenumbruchzeichen in die Zeichenkette eingebettet sind, wie im folgenden Beispiel:

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

Die Methode `writelines` ist jedoch verwirrend, weil sie nicht unbedingt mehrere Zeilen schreibt; sie nimmt eine Liste von Zeichenketten als Argument und schreibt sie nacheinander in das angegebene Datei-Objekt, ohne Zeilenumbrü-

che zwischen den Listenelementen einzufügen; nur wenn die Zeichenketten in der Liste Zeilenumbrüchen enthalten, kommen Zeilenumbrüche im Datei-Objekt hinzu; andernfalls werden sie aneinandergereiht. `writelines` ist damit die genaue Umkehrung von `readlines`, da sie auf die von `readlines` zurückgegebene Liste angewendet werden kann, um eine Datei zu schreiben, die identisch mit der Ausgangsdatei ist. Unter der Annahme, dass myfile.txt existiert und eine Textdatei ist, erzeugt das folgende Beispiel eine exakte Kopie von `myfile.txt` mit dem Namen `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()
```

Verwendung des Binärmodus

Wenn ihr alle Daten in einer Datei in ein einziges Byte-Objekt (partiell) einlesen und in den Speicher übertragen möchtet um sie als Byte-Sequenz behandeln zu können, könnt ihr die `read`-Methode verwenden. Ohne ein Argument liest sie die gesamte Datei ab der aktuellen Position ein und gibt die Daten als Bytes-Objekt zurück. Mit einem ganzzahligen Argument liest sie maximal diese Anzahl von Bytes und gibt ein Bytes-Objekt der angegebenen Größe zurück:

```
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()
```

Zeile 1
: öffnet eine Datei zum Lesen im Binärmodus

Zeile 2
: liest die ersten 16 Bytes als `head`-String

Zeile 3
: gibt den `head`-String aus

Zeile 5
: liest den Rest der Datei

Bemerkung: Dateien, die im Binärmodus geöffnet werden, arbeiten nur mit Bytes und nicht mit Zeichenketten. Um die Daten als Zeichenketten zu verwenden, müsst ihr alle Byte-Objekte in String-Objekte dekodieren. Dieser Punkt ist oft wichtig im Umgang mit Netzwerkprotokollen, wo sich Datenströme oft wie Dateien verhalten, aber als Bytes und nicht als Strings interpretiert werden müssen.

7.7.5 Eingebaute Module für Dateien

Die Python-Standardbibliothek enthält eine Reihe eingebauter Module, mit denen ihr Dateien managen könnt:

Modul	Beschreibung
`os.path`	führt allgemeine Pfadnamenmanipulationen durch
`pathlib`	manipuliert Pfadnamen
`fileinput`	iteriert über mehrere Eingabedateien
`filecmp`	vergleicht Dateien und Verzeichnisse
`tempfile`	erzeugt temporäre Dateien und Verzeichnisse
`glob`, `fnmatch`	verwenden UNIX-ähnlicher Pfad- und Dateinamensmuster
`linecache`	greift zufällig auf Textzeilen zu
`shutil`	führt Dateioperationen auf höherer Ebene aus
`mimetypes`	Zuordnung von Dateinamen zu MIME-Typen
`pickle`, `shelve`	aktivieren von Python-Objektserialisierung und -persistenz, s.a. (siehe auch) *Das pickle-Modul*
`csv`	liest und schreibt CSV-Dateien
`json`	JSON-Kodierer und -Dekodierer
`sqlite3`	bietet eine DB-API 2.0-Schnittstelle für SQLite-Datenbanken, s.a. *Das sqlite-Modul*
`xml`, `xml.parsers.expat`, `xml.dom`, `xml.sax`, `xml.etree.ElementTree`	liest und schreibt XML-Dateien, s.a. *Das xml-Modul*
`html.parser`, `html.entities`	Parsen von HTML und XHTML
`configparser`	liest und schreibt Windows-ähnliche Konfigurationsdateien (`.ini`)
`base64`, `binhex`, `binascii`, `quopri`, `uu`	Kodierung/Dekodierung von Dateien oder Streams
`struct`	liest und schreibt strukturierte Daten in und aus Dateien
`zlib`, `gzip`, `bz2`, `zipfile`, `tarfile`	für das Arbeiten mit Archivdateien und Komprimierungen

Siehe auch:

- pandas IO tools
- Beispiele für die Serialisierungsformate CSV, JSON, Excel, XML/HTML, YAML, TOML und Pickle.

7.8 None

Zusätzlich zu den Standardtypen wie *Zeichenketten* und *Zahlen* verfügt Python über einen speziellen Datentyp, der ein einziges spezielles Datenobjekt namens `None` definiert. Wie der Name schon sagt, wird `None` verwendet, um einen leeren Wert darzustellen. Er taucht in verschiedenen Formen in Python auf.

`None` ist in der alltäglichen Python-Programmierung oft als Platzhalter nützlich, um eine Datenstruktur zu kennzeichnen, an der irgendwann sinnvolle Daten gefunden werden können, auch wenn diese Daten noch nicht berechnet wurden.

Das Vorhandensein von `None` lässt sich leicht überprüfen, da es in Python nur eine Instanz von `None` gibt (alle Verweise auf `None` verweisen auf dasselbe Objekt), und `None` ist nur mit sich selbst identisch:

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

7.8.1 `None` ist *falsy*

In Python verlassen wir uns oft darauf, dass `None` *falsy* ist:

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

So können wir z.B. in einer *if-Anweisung* überprüfen, ob *Zeichenketten* leer sind:

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

7.8.2 `None` steht für Leere

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

7.8.3 Der Standardrückgabewert einer Funktion ist `None`

Eine Prozedur in Python ist beispielsweise nur eine Funktion, die nicht explizit einen Wert zurückgibt, was bedeutet, dass sie standardmäßig `None` zurückgibt:

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

KAPITEL 8

Input

Ihr könnt die Funktion `input()` verwenden, um Dateneingaben zu erhalten. Verwendet den Prompt-String, den ihr anzeigen möchtet, als Parameter für `input`:

```
>>> first_name = input("Vorname? ")
Vorname? Veit
>>> surname = input("Nachname? ")
Nachname? Schiele
>>> print(first_name, surname)
Veit Schiele
```

Dies ist ein recht einfacher Weg, um Dateneingaben zu erhalten. Der einzige Haken ist, dass die Eingabe als Zeichenkette eingeht. Wenn ihr also eine Zahl verwenden wollt, müsst ihr sie mit der Funktion `int` oder `float` umwandeln, z.B. für die Berechnung des Alters aus dem Geburtsjahr:

```
>>> import datetime
>>> currentDateTime = datetime.datetime.now()
>>> year = currentDateTime.year
>>> year_birth = input("Geburtsjahr? ")
Geburtsjahr? 1964
>>> age = year - int(year_birth)
>>> print("Alter:", age, "Jahre")
Alter: 58 Jahre
```

KAPITEL 9

Kontrollflüsse

Python verfügt über eine ganze Reihe von Strukturen zur Kontrolle der Code-Ausführung und des Programmablaufs, einschließlich gängiger Verzweigungen und Schleifen.

9.1 Boolesche Werte und Ausdrücke

In Python gibt es mehrere Möglichkeiten, boolesche Werte auszudrücken; die boolesche Konstante `False`, `0`, der Python-Typ *None* und leere Werte (z.B. die leere Liste `[]` oder die leere Zeichenkette `""`) werden alle als `False` betrachtet. Die boolesche Konstante `True` und alles andere wird als `True` betrachtet.

<, <=, ==, >, >=
: vergleicht Werte.

is, is not, in, not in
: überprüft die Identität.

and, not, or
: sind logischen Operatoren, mit denen die oben genannten Überprüfungen verknüpft werden können.

```
>>> 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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
>>> id(y)
4367574480
>>> id(z[0])
4375911432
```

Wenn `x` und `z[0]` die gleiche ID im Speicher haben, bedeutet das, dass wir an zwei Stellen auf dasselbe Objekt verweisen.

Am häufigsten werden `is` und `is not` in Verbindung mit *None* verwendet:

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

Der Python-Style-Guide in **PEP 8** besagt, dass ihr Identität verwenden solltet, um mit *None* zu vergleichen. Ihr solltet also niemals `x == None` verwenden, sondern stattdessen `x is None` eingeben.

Ihr solltet jedoch nie berechnete Fließkommazahlen miteinander vergleichen:

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

9.2 `if-elif-else`-Anweisung

Der Codeblock nach der ersten wahren Bedingung einer `if`- oder `elif`-Anweisung wird ausgeführt. Wenn keine der Bedingungen wahr ist, wird der Codeblock nach dem `else` ausgeführt:

```
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
```

Zeilen 5 und 8
: Die `elif`- und `else`-Klauseln sind optional, und es kann eine beliebige Anzahl von `elif`-Klauseln geben.

Zeilen 3, 4, 6, 7, 9 und 10
: Python verwendet Einrückungen, um Blöcke abzugrenzen. Es sind keine expliziten Begrenzungszeichen wie

Klammern oder geschweifte Klammern erforderlich. Jeder Block besteht aus einer oder mehreren Anweisungen, die durch Zeilenumbrüche getrennt sind. Alle diese Anweisungen müssen auf der gleichen Einrückungsebene stehen.

9.3 Schleifen

9.3.1 `while`-Schleife

Die `while`-Schleife wird so lange ausgeführt, wie die Bedingung (hier: `x > y`) wahr ist:

```
1  >>> x, y = 6, 3
2  >>> while x > y:
3  ...     x -= 1
4  ...     if x == 4:
5  ...         break
6  ...     print(x)
7  ...
8  5
```

Zeile 1
Dies ist eine Kurzschreibweise, wobei `x` den Wert `6` und `y` den Wert `3` erhält.

Zeilen 2–10
Dies ist die `while`-Schleife mit der Anweisung `x > y`, die wahr ist, solange `x` größer als `y` ist.

Zeile 3
`x` wird um `1` reduziert

Zeile 4
`if`-Bedingung, bei der `x` exakt `4` sein soll.

Zeile 5
`break` beendet die Schleife.

Zeilen 8 und 9
gibt die Ergebnisse der `while`-Schleife aus bevor die Ausführung mit `break` unterbrochen wurde.

```
1  >>> x, y = 6, 3
2  >>> while x > y:
3  ...     x -= 1
4  ...     if x == 4:
5  ...         continue
6  ...     print(x)
7  ...
8  5
9  3
```

Zeile 5
`continue` bricht die aktuelle Iteration der Schleife ab.

9.3.2 for-Schleife

Die for-Schleife ist einfach, aber mächtig, weil sie über einen beliebigen iterierbaren Typ, wie eine Liste oder ein Tupel, iterieren kann. Anders als in vielen anderen Sprachen iteriert die for-Schleife in Python über jedes Element in einer Sequenz (z.B. eine *Liste* oder ein *Tupel*), was sie eher zu einer foreach-Schleife macht. Die folgende Schleife verwendet den Modulo-Operator % als Bedingung für as erste Vorkommen einer ganzen Zahl, die durch 5 teilbar ist:

```
>>> 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"Erste gefundene Ganzzahl, die durch {d} teilbar ist: {i}")
...         break
...
Erste gefundene Ganzzahl, die durch 5 teilbar ist: 10
```

x wird nacheinander jeder Wert in der Liste zugewiesen. Wenn x keine ganze Zahl ist, wird der Rest dieser Iteration durch die continue-Anweisung abgebrochen. Die Ablaufsteuerung wird fortgesetzt, wobei x auf den nächsten Eintrag in der Liste gesetzt wird. Nachdem die erste passende ganze Zahl gefunden wurde, wird die Schleife mit der break-Anweisung beendet.

9.3.3 Schleifen mit einem Index

Ihr könnt in einer for-Schleife auch den Index ausgeben, z.B. mit 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

Üblicherweise wird eine Liste folgendermaßen generiert:

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

Anstatt eine leere Liste zu erstellen und jedes Element am Ende einzufügen, definiert ihr mit List Comprehensions einfach die Liste und ihren Inhalt gleichzeitig mit nur einer einzigen Code-Zeile:

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

Das allgemeine Format hierfür ist:

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

Jede List Comprehension in Python enthält drei Elemente:

EXPRESSION
: ist ein Aufruf einer Methode oder ein anderer gültiger Ausdruck, der einen Wert zurückgibt. Im obigen Beispiel ist der Ausdruck `i ** 2` das Quadrat des jeweiligen Mitgliedswertes.

MEMBER
: ist das Objekt oder der Wert in einem *ITERABLE*. Im obigen Beispiel ist der Wert `i`.

ITERABLE
: ist eine *Liste*, ein *Set*, ein Generator oder ein anderes Objekt, das seine Elemente einzeln zurückgeben kann. Im obigen Beispiel ist die Iterable `range(8)`.

Ihr könnt mit List Comprehensions auch optional Bedingungen verwenden, die üblicherweise am Ende des Ausdruck angehängt werden:

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

9.4 Exceptions

In diesem Abschnitt geht es um Ausnahmen, D.H. um Sprachfunktionen, die speziell ungewöhnliche Umstände während der Ausführung eines Programms behandeln. Die häufigste Ausnahme ist die Behandlung von Fehlern, aber sie können auch für viele andere Zwecke effektiv eingesetzt werden. Python bietet einen umfassenden Satz von Ausnahmen, und ihr könnt neue Ausnahmen für eure eigenen Zwecke definieren.

Der gesamte Exception-Mechanismus in Python ist *objektorientiert*: Eine Exception ist ein Objekt, das automatisch von Python-Funktionen mit einer `raise`-Anweisung erzeugt wird. Diese `raise`-Anweisung veranlasst die Ausführung des Python-Programms auf eine andere Art und Weise, als üblicherweise vorgesehen: Die aktuelle Aufrufkette wird nach einem Handler durchsucht, der die erzeugte Ausnahme behandeln kann. Wenn ein solcher Handler gefunden wird, wird er aufgerufen und kann auf das Ausnahmeobjekt zugreifen, um weitere Informationen zu erhalten. Wird kein geeigneter Exception-Handler gefunden, bricht das Programm mit einer Fehlermeldung ab.

Es ist möglich, verschiedene Arten von Ausnahmen zu erzeugen, um die tatsächliche Ursache des gemeldeten Fehlers oder außergewöhnlichen Umstandes zu reflektieren. Eine Übersicht über die Klassenhierarchie eingebauter Exceptions erhaltet ihr unter Exception hierarchy in der Python-Dokumentation. Jeder Ausnahmetyp ist eine Python-Klasse, die von ihrem übergeordneten Exception-Typ erbt. So ist z.B. (ZUM BEISPIEL ein `ZeroDivisionError` durch Vererbung auch ein `ArithmeticError`, eine `Exception` und auch eine `BaseException`. Diese Hierarchie ist gewollt: Die meisten Ausnahmen erben von `Exception`, und es wird dringend empfohlen, dass alle benutzerdefinierten Ausnahmen auch die Unterklasse von `Exception` und nicht von `BaseException` bilden:

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

Dies definiert ihr euren eigenen Ausnahmetyp, der vom Basistyp `Exception` erbt.

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

Eine Liste unterschiedlicher Datei-Arten wird definiert.

Schließlich werden Ausnahmen oder Fehler mit Hilfe der zusammengesetzten Anweisung `try-except-else-finally` abgefangen und behandelt. Jede Ausnahme, die nicht abgefangen wird, führt zur Beendigung des Programms.

```
7  for file in filenames:
8      try:
9          f = open(file, "r")
10         line = f.readline()
11         if line == "":
12             raise EmptyFileError(f"{file} is empty")
13     except OSError 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")
21         f.close()
```

Zeile 7
Wenn während der Ausführung der Anweisungen im `try`-Block ein `OSError` oder `EmptyFileError` auftritt, wird der zugehörige `except`-Block ausgeführt.

Zeile 9
Hier könnte ein `OSError` ausgelöst werden.

Zeile 12
Hier löst ihr den `EmptyFileError` aus.

Zeile 17
Die `else`-Klausel ist optional; sie wird ausgeführt, wenn im `try`-Block keine Ausnahme auftritt.

Bemerkung: In diesem Beispiel hätte stattdessen auch `continue`-Anweisungen in den `except`-Blöcken verwendet werden können.

Zeile 19
Die `finally`-Klausel ist optional; sie wird am Ende des Blocks ausgeführt, unabhängig davon, ob eine Ausnahme ausgelöst wurde oder nicht.

9.5 Kontextmanagement mit `with`

Eine rationellere Art, das Muster `try-except-finally` zu kapseln, ist die Verwendung des Schlüsselworts `with` und eines Kontextmanagers. Python definiert Kontextmanager für Dinge wie den Zugriff auf *Dateien* und eigene Kontextmanager. Ein Vorteil von Kontextmanagern ist, dass sie standardmäßige Bereinigungsaktionen definieren können, die immer ausgeführt werden, unabhängig davon, ob eine Ausnahme auftritt oder nicht.

Die folgende Auflistung zeigt das Öffnen und Lesen einer Datei unter Verwendung von `with` und einem Kontextmanager.

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

Hier wird ein Kontextmanager eingerichtet, der die Funktion `open` und den darauf folgenden Block umschließt. Die vordefinierte Aufräumaktion des Kontextmanagers schließt die Datei, auch wenn eine Ausnahme auftritt. Solange der

Ausdruck in der ersten Zeile ausgeführt wird, ohne eine Ausnahme auszulösen, wird die Datei immer geschlossen. Dieser Code ist äquivalent zu diesem Code:

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

KAPITEL 10

Funktionen

Die grundlegende Syntax für eine Python-Funktionsdefinition lautet

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

Wie bei *Kontrollströmen* verwendet Python Einrückungen, um die Funktion von der Funktionsdefinition abzugrenzen. Das folgende einfache Beispiel fügt den Code in eine Funktion ein, so dass ihr diese aufrufen könnt, um die Fakultät einer Zahl zu erhalten:

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

Zeile 2

Dies ist ein optionaler Dokumentationsstring, oder `docstring`. Ihr könnt seinen Wert erhalten, indem ihr `fact.__doc__` aufruft. Der Zweck von Docstrings ist es, das Verhalten einer Funktion und die Parameter, die sie annimmt, zu beschreiben, während Kommentare interne Informationen über die Funktionsweise des Codes dokumentieren sollen. Docstrings sind *Zeichenketten*, die unmittelbar auf die erste Zeile einer Funktionsdefinition folgen und normalerweise in dreifachen Anführungszeichen stehen, um mehrzeilige Beschreibungen zu ermöglichen. Bei mehrzeiligen Dokumentationsstrings ist es üblich, in der ersten Zeile eine Zusammenfassung der Funktion zu geben, dieser Zusammenfassung eine leere Zeile folgen zu lassen und mit dem Rest der Informationen zu enden.

Siehe auch:

- *Docstrings*

Zeile 7

Der Wert wird nach dem Aufruf der Funktion zurückgegeben. Ihr könnt auch Funktionen schreiben, die keine

Rückgabeanweisung haben und *None* zurückgeben, und wenn `return arg` ausgeführt wird, wird der Wert `arg` zurückgegeben.

Obwohl alle Python-Funktionen Werte zurückgeben, liegt es an euch, wie der Rückgabewert einer Funktion verwendet wird:

```
>>> fact(3)
6
>>> x = fact(3)
>>> x
6
```

Zeile 1
Der Rückgabewert ist nicht mit einer Variablen verknüpft.

Zeile 2
Der Wert der `fact`-Funktion wird nur im Interpreter ausgegeben.

Zeile 3
Der Rückgabewert ist mit der Variablen `x` verknüpft.

10.1 Parameter

Python bietet flexible Mechanismen zur Übergabe von Argumenten an Funktionen:

```
>>> 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}
```

Zeile 2
Funktionen werden mit Hilfe der `def`-Anweisung definiert.

Zeile 5
Die `return`-Anweisung wird von einer Funktion verwendet, um einen Wert urückzugeben. Dieser Wert kann von beliebigem Typ sein. Wird keine `return`-Anweisung gefunden, wird der Wert `None` von Python zurückgegeben.

Zeile 11
Funktionsargumente können entweder nach Position oder nach Name (Schlüsselwort) eingegeben werden. `z` und `y` werden in unserem Beispiel mit dem Namen angegeben.

Zeile 13
Funktionsparameter können mit Standardwerten definiert werden, die verwendet werden, wenn ein Funktionsaufruf sie auslässt.

Zeile 18
Es kann ein spezieller Parameter definiert werden, der alle zusätzlichen Positionsargumente in einem Funktionsaufruf in einem Tupel zusammenfasst.

Zeile 25
Ebenso kann ein spezieller Parameter definiert werden, der alle zusätzlichen Schlüsselwortargumente in einem Funktionsaufruf in einem Dictionary zusammenfasst.

10.1.1 Parameter

Optionen für Funktionsparameter

Die meisten Funktionen benötigen Parameter. Dabei bietet Python drei Optionen für die Definition von Funktionsparametern.

Positionsbezogene Parameter

Die einfachste Art, Parameter an eine Funktion in Python zu übergeben, ist die Übergabe an der Position. In der ersten Zeile der Funktion gebt ihr den Variablennamen für jeden Parameter an; wenn die Funktion aufgerufen wird, werden die im aufrufenden Code verwendeten Parameter den Parametervariablen der Funktion auf der Grundlage ihrer Reihenfolge zugeordnet. Die folgende Funktion berechnet `x` als Potenz von `y`:

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

Diese Methode setzt voraus, dass die Anzahl der vom aufrufenden Code verwendeten Parameter genau mit der Anzahl der Parameter in der Funktionsdefinition übereinstimmt; andernfalls wird eine Type-Error-Exception ausgelöst:

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

Funktionsparameter können Standardwerte haben, die ihr deklarieren könnt, indem ihr in der ersten Zeile der Funktionsdefinition einen Standardwert zuweist, etwa so:

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

Es können beliebig viele Parameter mit Standardwerten versehen werden wobei Parameter mit Standardwerten als letzte in der Parameterliste definiert werden müssen.

Die folgende Funktion berechnet `x` ebenfalls als Potenz von `y`. Wenn `y` jedoch nicht in einem Funktionsaufruf angegeben wird, wird der Standardwert `5` verwendet:

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

Wie sich das Standardargument auswirkt, können ihr im folgenden Beispiel sehen:

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

Parameternamen

ihr könnt auch Argumente an eine Funktion übergeben, indem ihr den Namen des entsprechenden Funktionsparameters und nicht dessen Position verwendet. Ähnlich dem vorherigen Beispiels könnt ihr Folgendes eingeben:

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

Da die Argumente für die Potenz im letzten Aufruf mit `x` und `y` benannt sind, ist ihre Reihenfolge irrelevant; die Argumente sind mit den gleichnamigen Parametern in der Definition der Potenz verknüpft, und man erhält `2^6` zurück. Diese Art der Argumentübergabe wird als Schlüsselwortübergabe bezeichnet. Die Übergabe von Schlüsselwörtern kann in Kombination mit den Standardargumenten von Python-Funktionen sehr nützlich sein, wenn ihr Funktionen mit einer großen Anzahl von möglichen Argumenten definiert, von denen die meisten gemeinsame Standardwerte haben.

Variable Anzahl von Argumenten

Python-Funktionen können auch so definiert werden, dass sie mit einer variablen Anzahl von Argumenten umgehen können. Dies ist auf zweierlei Arten möglich. Die eine Methode sammelt eine unbekannte Anzahl von Argumenten in einer *Liste*. Die andere Methode kann eine beliebige Anzahl von Argumenten, die mit einem Schlüsselwort übergeben wurde und die keinen entsprechend benannten Parameter in der Funktionsparameterliste hat, in einem *Dict* sammeln.

Bei einer unbestimmten Anzahl von Positionsargumenten bewirkt das Voranstellen eines `*` vor den endgültigen Parameternamen der Funktion, dass alle überschüssigen Nicht-Schlüsselwort-Argumente in einem Funktionsaufruf, D.H. die Positionsargumente, die keinem anderen Parameter zugewiesen sind, gesammelt und als Tupel dem angegebenen Parameter zugewiesen werden. Dies ist z.B. eine einfache Möglichkeit, eine Funktion zu implementieren, die den Mittelwert in einer Liste von Zahlen findet:

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

Nun könnt ihr das Verhalten der Funktion testen, z.B. mit:

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

Eine beliebige Anzahl von Schlüsselwortargumenten kann ebenfalls verarbeitet werden, wenn dem letzten Parameter in der Parameterliste das Präfix ** vorangestellt ist. Dann werden alle Argumente, die mit einem Schlüsselwort übergeben wurden, in einem *Dict* gesammelt. Der Schlüssel für jeden Eintrag im Dict ist das Schlüsselwort (Parametername) für das Argument. Der Wert dieses Eintrags ist das Argument selbst. Ein per Schlüsselwort übergebenes Argument ist in diesem Zusammenhang überflüssig, wenn das Schlüsselwort, mit dem es übergeben wurde, nicht mit einem der Parameternamen in der Funktionsdefinition übereinstimmt, z.B.:

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

Das Ausprobieren dieser Funktion zeigt, dass sie die Argumente addieren kann, die unter den Schlüsselwörtern `foo`, `bar` und `baz` übergeben werden, obwohl `foo`, `bar` und `baz` in der Funktionsdefinition keine Parameternamen sind:

```
>>> 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
```

Techniken zur Argumentübergabe mischen

Es ist möglich, alle Argumentübergabe-Möglichkeiten von Python-Funktionen gleichzeitig zu verwenden, obwohl dies verwirrend sein kann, wenn ihr es nicht sorgfältig macht. Dabei sollten die Positionsargumente an erster Stelle stehen, dann benannte Argumente, gefolgt von unbestimmten Positionsargumenten mit einem einfachen * und zuletzt unbestimmte Schlüsselwortargumente mit **.

Veränderliche Objekte als Argumente

Argumente werden per Objektreferenz übergeben. Der Parameter wird zu einem neuen Verweis auf das Objekt. Bei unveränderlichen Objekten wie *Tupel*, *Zeichenketten* und *Zahlen* hat das, was mit einem Parameter gemacht wird, keine Auswirkungen außerhalb der Funktion. Wenn ihr jedoch ein veränderliches Objekt übergeben, z.B. eine *Liste*, ein *Dict* oder eine Klasseninstanz, ändert jede Änderung des Objekts, worauf das Argument außerhalb der Funktion verweist. Die Neuzuweisung des Parameters hat keine Auswirkungen auf das 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])
```

Die Variable `x` wird nicht geändert, da sie unveränderlich ist. Stattdessen wird der Funktionsparameter `n` so gesetzt, dass er auf den neuen Wert 6 verweist. Bei `y` gibt es jedoch eine Änderung, weil die Liste, auf die sie verweist, geändert wurde.

10.1.2 Variablen

Lokale, nicht-lokale und globale Variablen

Hier kehren wir zur Definition von `fact` vom Anfang dieses *Funktionen*-Kapitels zurück:

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

Sowohl die Variablen `f` als auch `n` sind lokal für einen bestimmten Aufruf der Funktion `fact`; Änderungen an ihnen, die während der Ausführung der Funktion vorgenommen werden, haben keine Auswirkungen auf Variablen außerhalb der Funktion. Alle Variablen in der Parameterliste einer Funktion und alle Variablen, die innerhalb einer Funktion durch eine Zuweisung erzeugt werden, wie z.B. `f = 1`, sind für die Funktion lokal.

Ihr könnt eine Variable explizit zu einer globalen Variable machen, indem ihr sie mit der `global`-Anweisung deklariert, bevor sie verwendet wird. Globale Variablen können von der Funktion angesprochen und geändert werden. Sie existieren außerhalb der Funktion und können auch von anderen Funktionen, die sie als global deklarieren, oder von

Code, der sich nicht innerhalb einer Funktion befindet, aufgerufen und geändert werden. Hier ein Beispiel, das den Unterschied zwischen lokalen und globalen Variablen verdeutlicht:

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

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

In diesem Beispiel wird eine Funktion definiert, die `x` als globale Variable und `y` als lokale Variable behandelt und versucht, sowohl `x` als auch `y` zu ändern. Die Zuweisung an `x` innerhalb von `my_func` ist eine Zuweisung an die globale Variable `x`, die auch außerhalb von `my_func` existiert. Da `x` in `my_func` als global bezeichnet wird, ändert die Zuweisung diese globale Variable so, dass sie den Wert `1` anstelle des Wertes `3` beibehält. Dasselbe gilt jedoch nicht für `y`; die lokale Variable `y` innerhalb von `my_func` verweist zunächst auf denselben Wert wie die Variable `y` außerhalb von `my_func`, aber die Zuweisung bewirkt, dass `y` auf einen neuen Wert verweist, der für die Funktion `my_func` lokal ist.

Siehe auch:

- The global statement

Während `global` für eine Variable der obersten Ebene verwendet wird, bezieht sich `nonlocal` auf jede Variable in einem umschließenden Bereich.

Siehe auch:

- The nonlocal statement
- **PEP 3104**

10.1.3 Dekoratoren

Funktionen können auch als Argumente an andere Funktionen übergeben werden und die Ergebnisse anderer Funktionen zurückgegeben. So ist es z.B. möglich, eine Python-Funktion zu schreiben, die eine andere Funktion als Parameter annimmt, sie in eine andere Funktion einbettet, die etwas Ähnliches tut, und dann die neue Funktion zurückgibt. Diese neue Kombination kann dann anstelle der ursprünglichen Funktion verwendet werden:

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
>>> my_func = inf(my_func)
Information about my_func
>>> my_func("Hello", "Pythonistas!")
Execute function my_func with the argument(s)
('Hello', 'Pythonistas!')
```

Zeile 2
Die `inf`-Funktion gibt den Namen der Funktion, die sie umhüllt, aus.

Zeile 6
Wenn sie fertig ist, gibt die `inf`-Funktion die umhüllte Funktion zurück.

Ein Dekorator ist syntaktischer Zucker für diesen Prozess und ermöglicht euch, eine Funktion mit einem einzeiligen Zusatz in eine andere zu packen. Ihr erhaltet immer noch genau den gleichen Effekt wie beim vorherigen Code, aber der resultierende Code ist viel sauberer und leichter zu lesen. Die Verwendung eines Dekorators besteht ganz einfach aus zwei Teilen:

1. der Definition der Funktion, die andere Funktionen umhüllen oder *dekorieren* soll, und
2. der Verwendung eines `@`, gefolgt von dem Dekorator, unmittelbar bevor die umhüllte Funktion definiert wird.

Die Dekorfunktion sollte eine Funktion als Parameter annehmen und eine Funktion zurückgeben, wie folgt:

```
>>> @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!')
```

Zeile 1
Die Funktion `my_func` wird mit `@inf` dekoriert.

Zeile 7
Die umhüllte Funktion wird aufgerufen, nachdem die Dekorator-Funktion fertig ist.

functools

Das Python-`functools`-Modul ist für Funktionen höherer Ordnung gedacht, also Funktionen, die auf andere Funktionen wirken oder diese zurückgeben. Meist könnt ihr sie als Dekoratoren verwenden, so U.A.:

`functools.cache()`
Einfacher, leichtgewichtiger, Funktionscache ab Python 3.9, der manchmal auch *memoize* genannt wird. Er gibt dasselbe zurück wie `functools.lru_cache()` mit dem Parameter `maxsize=None`, wobei zusätzlich ein *Dictionaries* mit den Funktionsargumenten erstellt wird. Da alte Werte nie gelöscht werden müssen, ist diese Funktion dann auch kleiner und schneller. Ein Beispiel:

```
>>> from functools import cache
>>> @cache
... def factorial(n):
...     return n * factorial(n - 1) if n else 1
...
>>> factorial(8)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
7 40320
8 >>> factorial(10)
9 3628800
```

Zeile 6
Da es kein zuvor gespeichertes Ergebnis gibt, werden neun rekursive Aufrufe gemacht.

Zeile 8
macht nur zwei neue Aufrufe, da die anderen Ergebnisse aus dem Zwischenspeicher kommen.

`functools.wraps()`
Dieser Dekorator lässt die Wrapper-Funktion so, so wie die ursprüngliche Funktion aussehen mit ihren Namen und ihren Eigenschaften.

```
>>> 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'
```

Ohne `@wraps`-Dekorator wäre stattdessen Name und Docstring der `wrapper`-Methode zurückgegeben worden:

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

10.1.4 Lambda-Funktionen

In Python ist eine Lambda-Funktion eine anonyme Funktion, also eine Funktion, die ohne Namen deklariert wird. Es handelt sich um eine kleine und eingeschränkte Funktion, die nicht länger als eine Zeile ist. Wie eine normale Funktion kann eine Lambda-Funktion mehrere Argumente haben, aber nur einen Ausdruck, der ausgewertet und zurückgegeben wird.

Die Syntax einer Lambda-Funktion lautet:

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

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

Bemerkung: Es gibt in der Lambda-Funktion keine `return`-Anweisung. Der einzelne Ausdruck nach dem Doppelpunkt ist der Rückgabewert.

Im nächsten Beispiel wird eine lambda-Funktion innerhalb eines Funktionsaufrufs erstellt. Es gibt jedoch keine globale Variable, um die Werte der lambda-Funktion zu speichern:

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

In diesem Fall akzeptiert die Funktion `max()` das Argument `key`, das definiert, wie die Größe jedes Eintrags bestimmt werden soll. Mithilfe einer Lambda-Funktion, die jede Zeichenkette in eine ganze Zahl umwandelt, kann `max` die numerischen Werte vergleichen und so das erwartete Ergebnis ermitteln.

KAPITEL 11

Module

Module werden in Python verwendet, um größere Projekte zu organisieren. Die Python-Standardbibliothek ist in Module aufgeteilt, um sie überschaubarer zu machen. Ihr müsst euren eigenen Code zwar nicht in Modulen organisieren, aber wenn ihr umfangreichere Programme schreibt, oder Code, den ihr wiederverwenden möchten, solltet ihr dies tun.

11.1 Was ist ein Modul?

Ein Modul ist eine Datei, die Code enthält. Sie definiert eine Gruppe von Python-Funktionen oder anderen Objekten, und der Name des Moduls wird vom Namen der Datei abgeleitet. Module enthalten meist Python-Quellcode, können aber auch kompilierte C- oder C++-Objektdateien sein. Kompilierte Module und Python-Source-Module werden auf die gleiche Weise verwendet.

Module fassen nicht nur verwandte Python-Objekte zusammen, sondern helfen auch, Namenskonflikte zu vermeiden. Do könnt ihr für euer Programm ein Modul namens `mymodule` schreiben, das eine Funktion namens `my_func` definiert. Im selben Programm möchtet ihr vielleicht auch ein anderes Modul namens `othermodule` verwenden, das ebenfalls eine Funktion namens `my_func` definiert, aber etwas anderes tut als eure `my_func`-Funktion. Ohne Module wäre es unmöglich, zwei verschiedene Funktionen mit demelben Namen zu verwenden. Mit Modulen könnt ihr in eurem Hauptprogramm auf die Funktionen `mymodule.my_func` und `othermodule.my_func` verweisen. Die Verwendung der Modulnamen sorgt dafür, dass die beiden `my_func`-Funktionen nicht verwechselt werden, da Python SOG. Namespaces verwendet. Ein Namespace ist im Wesentlichen ein Wörterbuch mit Bezeichnungen für die dort zur Verfügung stehenden Funktionen, Klassen, Module USW..

Module werden auch verwendet, um Python selbst überschaubarer zu machen. Die meisten Standardfunktionen von Python sind nicht in den Kern der Sprache integriert, sondern werden über spezielle Module bereitgestellt, die ihr bei Bedarf laden könnt.

Siehe auch:

- Python Module Index

11.2 Erstellen von Modulen

Vermutlich der beste Weg, um etwas über Module zu lernen, ist das Erstellen eines eigenen Moduls. Hierzu erstellen wir eine Textdatei mit dem Namen `wc.py`, und geben in diese Textdatei den unten stehenden Python-Code ein. Wenn ihr *IDLE* verwendet, wählt *File → New Window* und beginnt mit der Eingabe.

Es ist einfach, eigene Module zu erstellen, die auf die gleiche Weise importiert und verwendet werden können wie die in Python eingebauten Bibliotheksmodule. Das folgende Beispiel ist ein einfaches Modul mit einer Funktion, die zur Eingabe eines Dateinamens auffordert und die Anzahl der in dieser Datei vorkommenden Wörter ermittelt.

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

Zeilen 1 und 5
Docstrings sind Standardmethoden zur Dokumentation von Modulen, Funktionen, Methoden und Klassen.

Zeile 10
`read` gibt eine Zeichenkette zurück, die alle Zeichen in einer Datei enthält, und `split` gibt eine Liste der Wörter einer Zeichenkette zurück, die anhand von Leerzeichen *aufgespalten* wurde.

Zeilen 25 und 26
Mit dieser `if`-Anweisung könnt ihr das Programm auf zweierlei Arten nutzen:

- zum Importieren in der Python-Shell oder einem anderen Python-Skript ist `__name__` der Dateiname:

```
>>> import wc
>>> wc.words_occur()
Enter the name of the file: README.rst
File README.rst has 332 words (191 are unique)
{'Schnelleinstieg': 1, ...}
```

Alternativ könnt ihr auch `words_occur` direkt importieren:

```
>>> from wc import words_occur
>>> words_occur()
Enter the name of the file: README.rst
File README.rst has 332 words (191 are unique)
{'Schnelleinstieg': 1, ...}
```

Ihr könnt den interaktiven Modus der Python-Shell oder von *IDLE* verwenden, um ein Modul während der Erstellung inkrementell zu testen. Wenn ihr jedoch euer Modul auf der Festplatte ändert, wird es durch die erneute Eingabe des Import-Befehls nicht erneut geladen. Zu diesem Zweck müsst ihr die Funktion `reload` aus dem importlib-Modul verwenden:

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

- als Skript wird es mit dem Namen `__main__` ausgeführt und die Funktion `words_occur()` aufgerufen:

```
$ python3 wc.py
Enter the name of the file: README.rst
File README.rst has 332 words (191 are unique)
{'Schnelleinstieg': 1, ...}
```

Speichert diesen Code zunächst in einem der Verzeichnisse des Modulsuchpfads, die in der Liste von `sys.path` zu finden ist. Als Dateinamensendung empfiehlt sich `.py`, da hierdurch die Datei als Python-Quellcode ausgewiesen wird.

Bemerkung: Die Liste von Verzeichnissen, die mit `sys.path` angezeigt wird, hängt von eurer Systemkonfiguration ab. Diese Liste von Verzeichnissen wird von Python in der Reihenfolge durchsucht, wenn eine Import-Anweisung ausgeführt wird. Das erste gefundene Modul, das die Importanforderung erfüllt, wird verwendet. Wenn es kein zutreffendes Modul in diesem Suchpfad gibt, wird ein `ImportError` ausgelöst.

Wenn ihr *IDLE* verwendet, könnt ihr euch den Suchpfad und die darin enthaltenen Module grafisch ansehen, indem ihr das Fenster *File* → *Path Browser* verwendet.

Die Variable `sys.path` wird mit dem Wert der Umgebungsvariablen `PYTHONPATH` initialisiert, falls diese existiert. Wenn ihr ein Python-Skript ausführt, wird in die `sys.path`-Variable für dieses Skript das Verzeichnis, in dem sich das Skript befindet, als erstes Element eingefügt, so dass ihr auf bequeme Weise feststellen könnt, wo sich das ausführende Python-Programm befindet.

11.3 Befehlszeilenargumente

Wollt ihr in unserem Beispiel den Dateinamen als Befehlszeilenargument übergeben, also mit

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

so könnt ihr dies einfach mit folgender Änderung unseres Scripts:

```
--- /home/docs/checkouts/readthedocs.org/user_builds/python-basics-tutorial-de/checkouts/
↪latest/docs/modules/wc.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/python-basics-tutorial-de/checkouts/
↪latest/docs/modules/wcargv.py
@@ -1,10 +1,12 @@
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
 """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()
```

sys.argv
: gibt eine Liste der Befehlszeilenargumente zurück, die an ein Python-Skript übergeben wurden. `argv[0]` ist der Skriptname.

.pop
: entfernt das Element an der angegebenen Position in der Liste und gibt es zurück. Wenn kein Index angegeben wird, entfernt `.pop()` das letzte Element in der Liste und gibt es zurück.

11.4 Das argparse-Modul

Ihr könnt ein Skript so konfigurieren, dass es sowohl Kommandozeilenoptionen als auch Argumente akzeptiert. Das argparse-Modul unterstützt beim Parsen verschiedener Argumenttypen und kann sogar Nachrichten erzeugen. Um das `argparse`-Modul zu verwenden, erstellt eine Instanz von `ArgumentParser`, füllt sie mit Argumenten und lest dann sowohl die optionalen als auch die Positionsargumente. Das folgende Beispiel veranschaulicht die Verwendung des Moduls:

```
--- /home/docs/checkouts/readthedocs.org/user_builds/python-basics-tutorial-de/checkouts/
↪latest/docs/modules/wc.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/python-basics-tutorial-de/checkouts/
↪latest/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()
```

Dieser Code erzeugt eine Instanz von `ArgumentParser` und fügt dann das Argument `filename` hinzu. Das `argparse`-Modul gibt ein Namespace-Objekt zurück, das die Argumente als Attribute enthält. Ihr könnt die Werte der Argumente mit der Punktnotation abrufen, in unserem Fall mit `args.filename`.

Ihr könnt das Skript nun aufrufen mit:

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

Zudem wird automatisch eine Hilfeoption `-h` oder `--help` erzeugt:

```
$ 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
```

KAPITEL 12

Programmbibliotheken

Mehrere *Module* können in einer Programmbibliothek zusammengefasst werden. Solche Bibliotheken ermöglichen euch, Module in Verzeichnissen und Unterverzeichnissen zu gruppieren und dann zu importieren und hierarchisch auf sie zu verweisen, indem ihr eine `package.subpackage.module`-Syntax verwendet. Dies erfordert nicht viel mehr als die Erstellung einer möglicherweise leeren Initialisierungsdatei für jedes Paket oder Unterpaket.

12.1 *„Batteries included“*

In Python kann eine Bibliothek aus mehreren Komponenten bestehen, einschließlich eingebauter Datentypen und Konstanten, die ohne eine Importanweisung verwendet werden können, wie z.B. *Zahlen* und *Listen*, sowie einiger eingebauter *Funktionen* und *Exceptions*. Der größte Teil der Bibliothek ist eine umfangreiche Sammlung von *Modulen*. Wenn ihr Python installiert habt, stehen euch auch verschiedene Bibliotheken zur Verfügung zum

- *Managen von Datentypen*
- *Ändern von Dateien*
- *Interagieren mit dem Betriebssystem*
- *Verwenden von Internet-Protokollen*
- *Entwickeln und Debuggen*

12.1.1 Managen von Datentypen

Die Standardbibliothek enthält natürlich Unterstützung für die in Python eingebauten Typen. Darüber hinaus gibt es in der Standardbibliothek drei Kategorien, die sich mit verschiedenen Datentypen befassen: Module für Strings, Datentypen und Zahlen.

String-Module

:

Modul	Beschreibung
`string`	vergleicht mit Konstanten wie `string.digits` oder `string.whitespace`
`re`	sucht und ersetzt Text mit regulären Ausdrücken
`struct`	interpretiert Bytes als gepackte Binärdaten
`difflib`	hilft beim Berechnen von Deltas, beim Auffinden von Unterschieden zwischen Zeichenketten oder Sequenzen und beim Erstellen von Patches und Diff-Dateien
`textwra`	umbricht und füllt Text, formatiert Text mit Zeilenumbrüchen oder Leerzeichen

Siehe auch:

- Manipulation von Zeichenketten mit pandas

Module für Datentypen

Modul	Beschreibung
`datetime`, `calendar`	Zeit- und Kalenderoperationen
`collections`	Container-Datentypen
`enum`	ermöglicht die Erstellung von Aufzählungsklassen, die symbolische Namen an konstante Werte binden
`array`	Effiziente Arrays von numerischen Werten
`sched`	Event-Scheduler
`queue`	Synchronisierte Queue-Klasse
`copy`	Flache und tiefe Kopieroperationen
`pprint`	druckt Python-Datenstrukturen „hübsch“ aus
`typing`	unterstützt die Kommentierung von Code mit Hinweisen auf die Typen von Objekten, insbesondere von Funktionsparametern und Rückgabewerten

Module für Zahlen

:

Modul	Beschreibung
`numbers`	für numerische abstrakte Basisklassen
`math`, `cmath`	für mathematische Funktionen für reelle und komplexe Zahlen
`decimal`	für dezimale Festkomma- und Gleitkomma-Arithmetik
`statistics`	für Funktionen zur Berechnung von mathematischen Statistiken
`fractions`	für rationale Zahlen
`random`	zum Erzeugen von Pseudozufallszahlen und -auswahlen sowie zum Mischen von Sequenzen
`itertools`	für Funktionen, die Iteratoren für effiziente Schleifen erzeugen
`functools`	für Funktionen höherer Ordnung und Operationen auf aufrufbaren Objekten
`operator`	für Standardoperatoren als Funktionen

12.1.2 Ändern von Dateien

:

Modul	Beschreibung
os.path	führt allgemeine Pfadnamenmanipulationen durch
pathlib	manipuliert Pfadnamen
fileinput	iteriert über mehrere Eingabedateien
filecmp	vergleicht Dateien und Verzeichnisse
tempfile	erzeugt temporäre Dateien und Verzeichnisse
glob, fnmatch	verwenden UNIX-ähnlicher Pfad- und Dateinamensmuster
linecache	greift zufällig auf Textzeilen zu
shutil	führt Dateioperationen auf höherer Ebene aus
mimetypes	Zuordnung von Dateinamen zu MIME-Typen
pickle, shelve	aktivieren von Python-Objektserialisierung und -persistenz, s.a. *Das pickle-Modul*
csv	liest und schreibt CSV-Dateien
json	JSON-Kodierer und -Dekodierer
sqlite3	bietet eine DB-API 2.0-Schnittstelle für SQLite-Datenbanken, s.a. *Das sqlite-Modul*
xml, xml.parsers.expat, xml.dom, xml.sax, xml.etree.ElementTree	liest und schreibt XML-Dateien, s.a. *Das xml-Modul*
html.parser, html.entities	Parsen von HTML und XHTML
configparser	liest und schreibt Windows-ähnliche Konfigurationsdateien (.ini)
base64, binhex, binascii, quopri, uu	Kodierung/Dekodierung von Dateien oder Streams
struct	liest und schreibt strukturierte Daten in und aus Dateien
zlib, gzip, bz2, zipfile, tarfile	für das Arbeiten mit Archivdateien und Komprimierungen

Siehe auch:

- pandas IO tools
- Beispiele für die Serialisierungsformate CSV, JSON, Excel, XML/HTML, YAML, TOML und Pickle.

12.1.3 Interagieren mit dem Betriebssystem

Modul	Beschreibung
os	Verschiedene Betriebssystemschnittstellen
platform	Zugang zu den Identifizierungsdaten der zugrunde liegenden Plattform
time	Zeitzugriff und Konvertierungen
io	Werkzeuge für die Arbeit mit Datenströmen
select	Warten auf I/O-Abschluss
optparse	Parser für Befehlszeilenoptionen
curses	Terminal-Handling für Zeichenzellen-Displays
getpass	Portable Passworteingabe
ctypes	bietet C-kompatible Datentypen
threading	High-Level Threading-Interface
multiprocessing	Prozessbasierte Threading-Schnittstelle
subprocess	Verwaltung von Unterprozessen

12.1.4 Verwenden von Internet-Protokollen

Modul	Beschreibung
socket, ssl	Low-Level-Netzwerkschnittstelle und SSL-Wrapper für Socket-Objekte
email	E-Mail- und MIME-Verarbeitungspaket
mailbox	Manipulation von Postfächern in verschiedenen Formaten
cgi, cgitb	Common Gateway Interface-Unterstützung
wsgiref	WSGI-Dienstprogramme und Referenzimplementierung
urllib.request, urllib.parse	Öffnen und Parsen von URLs
ftplib, poplib, imaplib, nntplib, smtplib, telnetlib	Clients für verschiedene Internetprotokolle
socketserver	Framework für Netzwerkserver
http.server	HTTP-Server
xmlrpc.client, xmlrpc.server	XML-RPC-Client und -Server

12.1.5 Entwickeln und Debuggen

Modul	Beschreibung
pydoc	Dokumentationsgenerator und Online-Hilfesystem
doctest	Beispiele aus Python-Docstrings testen
unittest	Framework für Unittests, s.a. *Unittest*
test.support	Utility-Funktionen für Tests
trace	verfolgt die Ausführung von Python-Anweisungen
pdb	Python-Debugger
logging	Protokollierungsfunktion für Python
timeit	misst die Ausführungszeit von kleinen Codeschnipseln
profile, cProfile	Python-Profiler
sys	Systemspezifische Parameter und Funktionen
gc	Funktionen des Python-Garbage-Collectors
inspect	inspiziert Objekte live
atexit	Exit-Handler
__future__	Zukünftige Statement-Definitionen
imp	erlaubt den Zugriff auf die Import-Interna
zipimport	importiert von Modulen aus Zip-Archiven
modulefinder	findet Module, die von einem Skript verwendet werden

12.2 Hinzufügen weiterer Python-Bibliotheken

Obwohl Pythons *„Batteries included“*-Philosophie bedeutet, dass ihr mit der Standardinstallation von Python bereits eine Menge machen könnt, wird unweigerlich die Situation kommen, in der ihr eine Funktionalität benötigt, die nicht in Python enthalten ist. Dieser Abschnitt gibt einen Überblick über die euch dann zur Verfügung stehenden Möglichkeiten.

Wenn ihr Glück habt, findet ihr die zusätzlich benötigte Funktionalität in einem Paket für euer Betriebssystem – mit einem ausführbaren Windows- oder macOS-Installationsprogramm oder einem Paket für eure Linux-Distribution.

Dies ist eine der einfachsten Möglichkeiten, eine Bibliothek zu eurer Python-Installation hinzuzufügen, da sich das Installationsprogramm oder euer Paketmanager um alle Details kümmert, um das Modul korrekt zu eurem System hinzuzufügen. Im Allgemeinen sind solche vorgefertigten Pakete jedoch nicht die Regel für Python-Software.

12.2.1 Installation von Python-Bibliotheken mit `pip` und `venv`

Wenn ihr ein Modul eines Drittanbieters benötigt, das nicht für eure Plattform vorgefertigt ist, müsst ihr euch an dessen Quelldistribution wenden. Dies bringt jedoch zwei Probleme mit sich:

1. Um die Quelldistribution zu installieren, müsst ihr sie finden und herunterladen.
2. Es werden bestimmte Python-Pfade und Berechtigungen eures Systems erwartet.

Python bietet *pip* als aktuelle Lösung für beide Probleme an. `pip` versucht, das Modul im *Python Package Index* (*PyPI*) zu finden, lädt es und alle Abhängigkeiten herunter und kümmert sich um die Installation. Die grundlegende Syntax von `pip` ist recht einfach: um z.B. die beliebte `requests`-Bibliothek von der Kommandozeile aus zu installieren, müsst ihr nur Folgendes tun:

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

Wenn ihr eine bestimmte Version eines Pakets angeben wollt, könnt ihr die Versionsnummern einfach anhängen:

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

oder

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

Installieren mit der `--user`-Option

Häufig werdet ihr ein Python-Paket jedoch nicht in der Hauptinstanz von Python installieren können oder wollen. Vielleicht benötigt ihr eine aktuellere Version einer Bibliothek, aber eine andere Anwendung benötigt noch eine ältere Version. Oder vielleicht habt ihr keine ausreichenden Administratorrechte, um das Standard-Python des Systems zu ändern. In solchen Fällen besteht eine Möglichkeit darin, die Bibliothek mit dem `--user`-Flag zu installieren: dieses installiert die Bibliothek in das Home-Verzeichnis, wo es dann allerdings auch nur für euch selbst nutzbar ist:

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

Siehe auch:

- Installing Python Modules

Virtuelle Umgebungen

Es gibt jedoch noch eine bessere Möglichkeit, wenn ihr die Installation von Bibliotheken im Python-System vermeiden wollt. Diese Option wird als *virtuelle Umgebung* (`virtualenv`) bezeichnet. Sie ist eine in sich geschlossene Verzeichnisstruktur, die sowohl eine Installation von Python als auch die zusätzlichen Pakete enthält. Da die gesamte Python-Umgebung in der virtuellen Umgebung enthalten ist, können die dort installierten Bibliotheken und Module nicht mit denen im Hauptsystem oder in anderen virtuellen Umgebungen kollidieren, so dass verschiedene Anwendungen unterschiedliche Versionen von Python und seinen Paketen verwenden können. Das Erstellen und Verwenden einer virtuellen Umgebung erfolgt in zwei Schritten:

1. Zuerst erstellen wir die Umgebung:

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

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

   Hiermit wird die Umgebung mit Python und *pip* in einem Verzeichnis namens `myenv` erstellt.

2. Anschließend könnt ihr diese Umgebung aktivieren, sodass beim nächsten Aufruf von `python` das Python aus eurer neuen Umgebung verwendet wird:

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

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

3. Python-Pakete nur für diese virtuelle Umgebung installieren:

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

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

4. Wenn ihr eure Arbeit an diesem Projekt beenden wollt, könnt ihr die virtuelle Umgebung wieder deaktivieren mit

```
(myenv) $ deactivate
```

```
(myenv) > deactivate
```

Siehe auch:

- Virtual Environments and Packages

PyPI

Der *Python Package Index* (*PyPI*) ist der Standard-Paket-Index, jedoch keineswegs das einzige Repository für Python-Code. Ihr könnt ihn direkt unter *pypi.org* aufrufen und nach Paketen suchen oder die Pakete nach Kategorien filtern.

12.3 Pakete und Programme

12.3.1 wheels

Das derzeitige Standardformat zur Verteilung von Python-Bibliotheken und -Programmen ist die Verwendung von *wheels*. wheels wurden entwickelt, um die Installation von Python-Code zuverlässiger zu machen und die Verwaltung von Abhängigkeiten zu erleichtern. Die Details zur Erstellung von wheels würden jedoch den Rahmen dieses Abschnitts sprengen, aber alle Details zu den Anforderungen und dem Prozess zur Erstellung von wheels findet ihr in *Verteilungspaket erstellen*.

Siehe auch:

- Pradyun Gedam: Thoughts on the Python packaging ecosystem
- Python Package Build Tools

12.3.2 py2exe und py2app

py2exe erstellt eigenständige Windows-Programme und py2app dasselbe für macOS. In beiden Fällen handelt es sich um einzelne ausführbare Dateien, die auch auf Rechnern laufen können, auf denen Python nicht installiert ist. In vielerlei Hinsicht sind jedoch eigenständige ausführbare Dateien nicht ideal, da sie tendenziell größer und weniger flexibel sind als native Python-Anwendungen, aber in manchen Situationen können sie auch die beste oder einzige Lösung sein.

12.3.3 freeze

Auch das `freeze`-Tool erstellt ein ausführbares Python-Programm, das auf Rechnern läuft, auf denen Python nicht installiert ist. Wenn ihr das `freeze`-Tool verwenden möchtet, müsst ihr wahrscheinlich den Python-Quellcode herunterladen.

Beim *Einfrieren* eines Python-Programms werden C-Dateien erstellt , die dann mit einem C-Compiler kompiliert und gelinkt werden, den ihr auf eurem System installiert haben müsst. Die so eingefrorene Anwendung läuft nur auf Plattformen, für die der verwendete C-Compiler seine ausführbaren Dateien bereitstellt.

Siehe auch:

- Tools/freeze

12.3.4 PyInstaller and PyOxidizer

PyInstaller und PyOxidizer bündeln eine Python-Anwendung und alle ihre Abhängigkeiten in einem einzigen Paket.

12.3.5 Briefcase

Briefcase ist ein Werkzeug zur Konvertierung eines Python-Projekts in eine eigenständige native Anwendung für Mac, Windows, Linux, iPhone/iPad und Android.

12.4 Verteilungspaket erstellen

Verteilungspakete (engl.: *Distribution Packages* sind Archive, die in einen Paket-Index wie z.B. *pypi.org* hochgeladen und mit *pip* installiert werden können.

Einige der folgenden Befehle erfordern eine neue Version von pip, sodass ihr sicherstellen solltet, dass ihr die neueste Version installiert habt:

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

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

12.4.1 Struktur

Ein minimales Distribution Package kann z.B. so aussehen:

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

12.4.2 pyproject.toml

PEP 517 und **PEP 518** brachten erweiterbare Build-Backends, isolierte Builds und *pyproject.toml* im TOML-Format.

`pyproject.toml` teilt U.A. *pip* und *build* mit, welches *Backend*-Werkzeug verwendet werden soll, um Distributionspakete für euer Projekt zu erstellen. Ihr könnt aus einer Reihe von Backends wählen, wobei dieses Tutorial standardmäßig `hatchling` verwendet.

Eine minimale und dennoch funktionale `dataprep/pyproject.toml`-Datei sieht dann z.B. so aus:

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

build-system
: definiert einen Abschnitt, der das Build-System beschreibt

requires
: definiert eine Liste von Abhängigkeiten, die installiert sein müssen, damit das Build-System funktioniert, in unserem Fall `hatchling`.

Bemerkung: Versionsnummern von Abhängigkeiten sollten üblicherweise jedoch nicht hier festgeschrieben werden sondern in der requirements.txt-Datei.

---

build-backend
: identifiziert den Einstiegspunkt für das Build-Backend-Objekt als gepunkteten Pfad. Das `hatchling`-Backend-Objekt ist unter `hatchling.build` verfügbar.

Bemerkung: Für Python-Pakete, die *binäre Erweiterungen* mit `Cython`, `C`-, `C++`-, `Fortran`- oder `Rust` enthalten, ist das *hatchling*-Backend jedoch nicht geeignet. Hier sollte eines der folgenden Backends verwendet werden:

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

Doch damit nicht genug – es gibt noch weitere Backends:

- *Flit*
- *whey*
- *poetry*

- *pybind11*
- *meson-python*

Siehe auch:

- pypackaging-native

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

Metadaten

In `pyproject.toml` könnt ihr auch Metadaten zu eurem Paket angeben, wie z.B.:

```
[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
: ist der Distributionsname eures Pakets. Dies kann ein beliebiger Name sein, solange er nur Buchstaben, Zahlen, `.`, `_` und `-` enthält. Er sollte auch nicht bereits auf dem *Python Package Index* (*PyPI*) vergeben sein.

version
: ist die Version des Pakets.

  In unserem Beispiel ist die Versionsnummer statisch gesetzt worden. Es gibt jedoch auch die Möglichkeit, die Version dynamisch anzugeben, z.B. durch eine Datei:

```
[project]
...
dynamic = ["version"]

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

Das Standardmuster sucht nach einer Variablen namens `__version__` oder `VERSION`, die die Version enthält, optional mit dem vorangestellten Kleinbuchstaben `v`. Dabei basiert das Standardschema auf **PEP 440**.

Wenn dies nicht der Art entspricht, wie ihr Versionen speichern wollt, könnt ihr mit der Option `pattern` auch einen anderen regulären Ausdruck definieren.

Siehe auch:

- Calendar Versioning
- ZeroVer

Es gibt jedoch noch weitere Versionsschema-Plugins, wie z.B. hatch-semver für Semantic Versioning.

Mit dem Version-Source-Plugin hatch-vcs könnt ihr auch Git-Tags verwenden:

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

Auch das setuptools-Backend erlaubt dynamische Versionierung:

```
[build-system]
requires = ["setuptools>=61.0", "setuptools-scm"]
build-backend = "setuptools.build_meta"

[project]
...
dynamic = ["version"]

[tool.setuptools.dynamic]
version = {attr = "dataprep.VERSION"}
```

Siehe auch:

- Configuring setuptools using pyproject.toml files: Dynamic Metadata

`authors`
: wird verwendet, um die Autoren des Pakets anahnd ihrer Namen und E-Mail-Adressen zu identifizieren.

  Ihr könnt auch `maintainers` im selben Format auflisten.

`description`
: ist eine kurze Zusammenfassung des Pakets, die aus einem Satz besteht.

`readme`
: ist ein Pfad zu einer Datei, die eine detaillierte Beschreibung des Pakets enthält. Diese wird auf der Paketdetailseite auf *Python Package Index* (*PyPI*) angezeigt. In diesem Fall wird die Beschreibung aus `README.rst` geladen.

`requires-python`
: gibt die Versionen von Python an, die von eurem Projekt unterstützt werden. Dabei werden Installationsprogramme wie *pip* ältere Versionen von Paketen durchsuchen, bis sie eines finden, das eine passende Python-Version hat.

`classifiers`
: gibt dem *Python Package Index* (*PyPI*) und *pip* einige zusätzliche Metadaten über euer Paket. In diesem Fall ist das Paket nur mit Python 3 kompatibel, steht unter der BSD-Lizenz und ist OS-unabhängig. Ihr solltet immer

zumindest die Versionen von Python angeben, unter denen euer Paket läuft, unter welcher Lizenz euer Paket verfügbar ist und auf welchen Betriebssystemen euer Paket läuft. Eine vollständige Liste der Klassifizierer findet ihr unter https://pypi.org/classifiers/.

Außerdem haben sie eine nützliche Zusatzfunktion: Um zu verhindern, dass ein Paket zu *PyPI* hochgeladen wird, verwendet den speziellen Klassifikator `"Private :: Do Not Upload"`. *PyPI* wird immer Pakete ablehnen, deren Klassifizierer mit `"Private ::"` beginnt.

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

Siehe auch:

PEP 631

urls
lässt euch eine beliebige Anzahl von zusätzlichen Links auflisten, die auf dem *Python Package Index* (*PyPI*) angezeigt werden. Im Allgemeinen könnte dies zum Quellcode, zur Dokumentation, zu Aufgabenverwaltungen usw. führen.

Siehe auch:

- Declaring project metadata
- **PEP 345**

Optionale Abhängigkeiten

project.optional-dependencies
erlaubt euch, optionale Abhängigkeiten für euer Paket anzugeben. Dabei könnt ihr auch zwischen verschiedenen Sets unterscheiden:

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

Auch rekursive optionale Abhängigkeiten sind mit pip 21.2 möglich. So könnt ihr beispielsweise für `dev` neben `pre-commit` auch alle Abhängigkeiten aus `docs` und `test` übernehmen:

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

Ihr könnt diese optionalen Abhängigkeiten installieren, z.B. mit:

```
$ cd /PATH/TO/YOUR/DISTRIBUTION_PACKAGE
$ python3 -m venv .
$ source bin/activate
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
$ 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

Wenn ihr ein neues Paket erstellt, solltet ihr kein flaches sondern das `src`-Layout verwenden, das auch in Packaging Python Projects der *PyPA* empfohlen wird. Ein wesentlicher Vorteil dieses Layouts ist, dass Tests mit der installierten Version eures Pakets und nicht mit den Dateien in eurem Arbeitsverzeichnis ausgeführt werden.

Siehe auch:

- Hynek Schlawack: Testing & Packaging

Bemerkung: In Python 3.11 kann mit `PYTHONSAFEPATH` sichergestellt werden, dass die installierten Pakete zuerst verwendet werden.

`dataprep`
: ist das Verzeichnis, das die Python-Dateien enthält. Der Name sollte mit dem Projektnamen übereinstimmen um die Konfiguration zu vereinfachen und für diejenigen, die das Paket installieren, besser erkennbar zu sein.

`__init__.py`
: ist erforderlich, um das Verzeichnis als Paket zu importieren. Die Datei sollte leer sein.

`loaders.py`
: ist ein Beispiel für ein Modul innerhalb des Pakets, das die Logik (Funktionen, Klassen, Konstanten, ETC. (et cetera)) eures Pakets enthalten könnte.

12.4.4 Andere Dateien

`CONTRIBUTORS.rst`

Siehe auch:

- All contributors

`LICENSE`

Ausführliche Informationen hierzu findet ihr im Abschnitt Lizenzieren.

README.rst

Diese Datei teilt denjenigen, die sich für das Paket interessieren, in kurzer Form mit, wie sie es nutzen können.

Siehe auch:

- Make a README
- readme.so

Wenn ihr das Dokument in *reStructuredText* schreibt, könnt ihr die Inhalte auch als ausführliche Beschreibung in euer Paket übernehmen:

```
[project]
readme = "README.rst"
```

Zudem könnt ihr sie dann auch in eure *Sphinx-Dokumentation* mit `.. include:: ../../README.rst` übernehmen.

CHANGELOG.rst

Siehe auch:

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

12.4.5 Historische oder für binäre Erweiterungen benötigte Dateien

Bevor die mit **PEP 518** eingeführte `pyproject.toml`-Datei zum Standard wurde, benötigte `setuptools` `setup.py`, `setup.cfg` und `MANIFEST.in`. Heute werden die Dateien jedoch bestenfalls noch für *binäre Erweiterungen* benötigt.

Wenn ihr diese Dateien in euren Paketen ersetzen wollt, könnt ihr dies mit `hatch new --init` oder ini2toml.

setup.py

Eine minimale und dennoch funktionale `dataprep/setup.py` kann z.B. so aussehen:

```
from Cython.Build import cythonize
from setuptools import find_packages, setup

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

package_dir verweist auf das Verzeichnis `src`, in dem sich ein oder mehrere Pakete befinden können. Anschließend könnt ihr mit setuptools's find_packages() alle Pakete in diesem Verzeichnis finden.

Bemerkung: `find_packages()` ohne `src/`-Verzeichnis würde alle Verzeichnisse mit einer `__init__.py`-Datei paketieren, also auch `tests/`-Verzeichnisse.

---

setup.cfg

Diese Datei wird nicht mehr benötigt, zumindest nicht für die Paketierung. `wheel` sammelt heutzutage alle erforderlichen Lizenzdateien automatisch und `setuptools` kann mit dem `options`-Keyword-Argument universelle `wheel`-Pakete bauen, z.B. `dataprep-0.1.0-py3-none-any.whl`.

MANIFEST.in

Die Datei enthält alle Dateien und Verzeichnisse, die nicht bereits mit `packages` oder `py_module` erfasst werden. Sie kann z.B. so aussehen: `dataprep/MANIFEST.in`:

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

Weitere Anweisungen in `Manifest.in` findet ihr in MANIFEST.in commands.

Bemerkung: Häufig wird die Aktualisierung der `Manifest.in`-Datei vergessen. Um dies zu vermeiden, könnt ihr check-manifest in einem Git pre-commit Hook verwenden.

Bemerkung: Wenn Dateien und Verzeichnisse aus `MANIFEST.in` auch installiert werden sollen, z.B. wenn es sich um laufzeitrelevante Daten handelt, könnt ihr dies mit `include_package_data=True` in eurem `setup()`-Aufruf angeben.

12.4.6 Build

Der nächste Schritt besteht darin, Distributionspakete für das Paket zu erstellen. Dies sind Archive, die in den *Python Package Index* (*PyPI*) hochgeladen und von *pip* installiert werden können.

Stellt sicher, dass ihr die neueste Version von `build` installiert habt:

Führt nun den Befehl in demselben Verzeichnis aus, in dem sich `pyproject.toml` befindet:

```
$ 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
```

Die zweite Zeile stellt sicher, dass ein sauberes Build ohne Artefakte früherer Builds erstellt wird. Die dritte Zeile sollte eine Menge Text ausgeben und nach Abschluss zwei Dateien im `dist`-Verzeichnis erzeugen:

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

dataprep-0.1.0-py3-none-any.whl
: ist eine Build-Distribution. Neuere pip-Versionen installieren bevorzugt Build-Distributionen, greifen aber bei Bedarf auf Source-Distributionen zurück. Ihr solltet immer eine Source-Distribution hochladen und Build-Distributionen für die Plattformen bereitstellen, mit denen euer Projekt kompatibel ist. In diesem Fall ist unser Beispielpaket mit Python auf jeder Plattform kompatibel, so dass nur eine Build-Distribution benötigt wird:

  dataprep
  : ist der normalisierte Paketname

  0.1.0
  : ist die Version des Distrubitionspakets

  py3
  : gibt die Python-Version und ggf. die C-ABI an

  none
  : gibt an, ob das *Wheel*-Paket für jedes oder nur spezifische OS geeignet ist

  any
  : any eignet sich für jede Prozessorarchitektur, `x86_64` hingegen nur für Chips mit dem x86-Befehlssatz und einer 64-Bit-Architektur

dataprep-0.1.0.tar.gz
: ist eine *Source Distribution*.

Siehe auch:

Die Referenz für die Dateinamen findet ihr in **PEP 427**.

Weitere Infos zu Source-Distributionen erhaltet ihr in Creating a Source Distribution. und **PEP 376**.

12.4.7 Testen

```
$ mkdir test_env
$ cd test_env
$ python3 -m venv .
$ . 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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
> 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:

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

Alternativ könnt ihr das Paket auch installieren:

```
$ 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
```

Anschließend könnt ihr Python aufrufen und euer `loaders`-Modul importieren:

```
from dataprep import loaders
```

Bemerkung: Es gibt immer noch viele Anleitungen, die einen Schritt zum Aufruf der `setup.py` enthalten, z.B. `python setup.py sdist`. Dies wird jedoch heutzutage von Teilen der Python Packaging Authority (PyPA) als Anti-Pattern angesehen.

12.5 Vorlagen

Mit Cookiecutter lassen sich Dateistrukturen erstellen, die u.a. das Erstellen von Python-Paketen deutlich vereinfachen.

Siehe auch:

- Copier

12.5.1 Übersicht

Ein minimales CookieCutter-Template sieht so aus:

```
cookiecutter-namespace-template/
├── {{ cookiecutter.project_name }}/
│   └── ...
└── cookiecutter.json
```

Zeile 1
Dies ist die Projektvorlage

Zeile 4
In dieser Datei sind die Eingabeaufforderungen und Standardwerte festgelegt. Die `cookiecutter.json`-Datei kann beispielsweise so aussehen:

```
{
  "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"]
}
```

Darüberhinaus können beliebige Verzeichnisse und Dateien angelegt werden.

Als Ergebnis erhaltet ihr dann folgende Dateistruktur:

```
my.package/
└── ...
```

12.5.2 CookieCutter-Features

- Cross-platform: Windows, Mac und Linux werden unterstützt
- Funktioniert mit Python 3.7, 3.8, 3.9 und 3.10
- Die Projektvorlagen können für jede Programmiersprache und jedes Markup-Format erstellt werden: Python, JavaScript, Ruby, ReST, CSS, HTML. Es können auch mehrere Sprachen im selben Template verwendet werden.
- Templates lassen sich einfach im Terminal anpassen:

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

- Ihr könnt auch lokale Templates verwenden:

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

- Alternativ könnt ihr CookieCutter auch mit Python verwenden:

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

- Verzeichnis- und Dateinamen können Vorlagen zugewiesen werden, z.B.:

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

- Die Verschachtelungstiefe ist unbegrenzt
- Das Templating basiert auf Jinja
- Ihr könnt eure Template-Variablen einfach in einer `cookiecutter.json`-Datei speichern, beispielsweise:

```
{
  "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"]
}
```

- Ihr könnt die Werte auch für mehrere Vorlagen hinterlegen in `~/cookiecutterrc`:

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

- CookieCutter-Templates, die aus einem Repository geladen wurden, werden üblicherweise in `~/.cookiecutters/` gespeichert. Anschließend können sie direkt über ihren Verzeichnisnamen referenziert werden, also z.B. mit:

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

12.5.3 Verfügbare Templates

Python

cookiecutter-namespace-template
: Namespace-Template für Python-Pakete

cookiecutter-pypackage
: Template für Python-Pakete

cookiecutter-pytest-plugin
: Minimales Cookiecutter-Template zum Erstellen von Pytest-Plugins

cookiecutter-pylibrary
: Umfangreiche Vorlage für Python-Pakete mit Unterstützung für Tests und Deployments (C-Extension-Support u.a. für cffi und Cython, Test-Unterstützung für Tox, Pytest, Travis-CI, Coveralls, Codacy, und Code Climate, Dokumentation mit Sphinx, Packaging-Checks u.a. mit scrutinizer, Isort etc.

cookiecutter-python-cli
: Template zum Erstellen einer Python-CLI-Anwendung mit Click

widget-cookiecutter
: Template zum Erstellen von Jupyter-Widgets

Ansible

cookiecutter-ansible-role-ci
: Vorlage für Ansible-Roles

C

bootstrap.c
: Template für in C mit Autotools geschriebene Projekte

cookiecutter-avr
: Template für die AVR-Entwicklung

C++

BoilerplatePP
: cmake-Template mit Unit Tests für C++-Projekte

Scala

cookiecutter-scala
: Vorlage für ein *Hello world*-Beispiel mit ein paar wenigen Bibliotheken

cookiecutter-scala-spark
: Template für eine Apache-Spark-Anwendung

LaTeX/XeTeX

pandoc-talk
: Template für Präsentation mit pandoc und XeTeX

12.5.4 Installation

Voraussetzungen

- Python-Interpreter
- Pfad zum Basisverzeichnis für euere Python-Pakete

  Stellt sicher, dass sich euer `bin`-Verzeichnis im Pfad befindet. In der Regel ist dies `~/.local/` für Linux und macOS oder `%APPDATA%\Python` unter Windows. Weitere Infos findet ihr in site.USER_BASE.

  Für Bash könnt ihr den Pfad in eurer `~/.bash_profile` angeben:

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

  und anschließend die Datei einlesen mit:

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

  Stellt sicher, dass das Verzeichnis, in dem CookieCutter installiert wird, sich in eurem `Path` befindet, damit ihr es direkt aufrufen könnt. Sucht dazu auf eurem Computer nach *Environment Variables* und fügt dieses Verzeichnis zu `Path` hinzu, also z.B. `%APPDATA%\Python\Python3x\Scripts`. Anschließend müsst ihr vermutlich die Session neu starten um die Umgebungsvariablen nutzen zu können.

  Siehe auch:

  Configuring Python

Installation

```
$ python3 -m venv cookiecutter_env
$ cd !$
cd cookiecutter_env
$ source bin/activate
$ python -m pip install cookiecutter
```

12.5.5 Fortgeschrittene Nutzung

Hooks

Ihr könnt sog. Pre- oder Post-Generate-Hooks schreiben, di entweder vor oder nach dem Generieren der Vorlage in den Ablauf eingehängt werden. Dabei werden die Jinja-Template-Variablen in die Skripte integriert, z.B.:

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

In einem Pre-Generate-Hook können z.B. Variablen validiert werden:

```
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)
```

Konfiguration

Wenn ihr Cookiecutter häufig verwendet, empfiehlt sich eine eigene Konfigurationsdatei: `~/cookiecutterrc`, z.B.:

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

Wiederholung

Beim Aufruf von `cookiecutter` wird eine `json`-Datei angelegt in `/.cookiecutter_replay/`, z.B. `~/.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"}}
```

Wenn ihr dies diese Informationen erneut verwenden wollt ohne diese erneut in der Kommandozeile bestätigen zu müssen, könnt ihr z.B. einfach folgendes eingeben:

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

Alternativ kann auch die Python-API verwendet werden:

```
from cookiecutter.main import cookiecutter

cookiecutter("gh:veit/cookiecutter-namespace-template", replay=True)
```

Diese Funktion ist hilfreich, wenn ihr z.B. ein Projekt aus einer aktualisierten Vorlage erstellen wollt.

Auswahlvariablen

Auswahlvariablen bieten verschiedene Möglichkeiten beim Erstellen eines Projekts. Abhängig von der Wahl des Benutzers rendert die Vorlage diese anders, z.B. wenn in der `cookiecutter.json`-Datei folgende Auswahl angeboten wird:

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

Dies wird dann ausgewertet 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 %}
```

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

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

12.5.6 cruft

Ein Problem mit cookiecutter-Vorlagen besteht darin, dass Projekte, die auf älteren Versionen der Vorlage basieren, veralten, wenn sich nur die Vorlage im Laufe der Zeit den sich ändeernden Anforderungen angepasst wird. cruft versucht, die Übernahme von Änderungen im Git-Repository des *Cookiecutter-Templates* in daraus abgeleitete Projekte zu vereinfachen.

Die wesentlichen Features von cruft sind:

- Mit `cruft check` könnt ihr schnell überprüfen, ob ein Projekt die neueste Version einer Vorlage verwendet. Diese Prüfung kann auch leicht in CI-Pipelines integriert werden, um sicherzustellen, dass eure Projekte synchron sind.
- cruft automatisiert auch die Aktualisierung der Projekte aus den cookiecutter-Vorlagen.

Installation

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

Ein neues Projekt erstellen

Um ein neues Projekt mit cruft zu erstellen, könnt ihr `cruft create` *`PROJECT_URL`* auf der Kommandozeile ausführen, z.B.:

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

cruft verwendet dabei *Cookiecutter* und der einzige Unterschied in der resultierenden Ausgabe ist eine `.cruft.json`-Datei, die den Git-Hash der verwendeten Vorlage sowie die angegebenen Parameter enthält.

Tipp: Bestimmte Dateien eignen sich selten zum Aktualisieren, z.B. Testfälle oder `__init__`-Dateien. Ihr könnt cruft anweisen, die Aktualisierung dieser Dateien in einem Projekt immer zu überspringen, indem ihr das Projekt mit den Argumenten `--skip vsc__init__.py --skip tests` erzeugt oder sie manuell zu einem Skip-Abschnitt in eurer `.cruft.json`-Datei hinzufügt:

```
{
  "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
}
```

Ein Projekt aktualisieren

Um ein bestehendes Projekt zu aktualisieren, das mit cruft erstellt wurde, könnt ihr `cruft update` im Stammverzeichnis des Projekts ausführen. Wenn es Aktualisierungen gibt, wird cruft euch zunächst bitten, diese zu überprüfen. Wenn ihr die Änderungen akzeptiert, wird cruft sie auf euer Projekt anwenden und die Datei `.cruft.json` aktualisieren.

Ein Projekt überprüfen

Um festzustellen, ob ein Projekt eine Vorlagenaktualisierung verpasst hat, könnt ihr ganz einfach, `cruft check` aufrufen. Wenn das Projekt veraltet ist, wird ein Fehler und der `Exit-Code 1` zurückgegeben. `cruft check` kann auch zu pre-commit-Framework und CI-Pipelines hinzugefügt werden, um sicherzustellen, dass Projekte nicht ungewollt veralten.

Ein bestehendes Projekt verknüpfen

Wenn ihr ein bestehendes Projekt habt, das ihr in der Vergangenheit mit Cookiecutter direkt aus einer Vorlage erstellt habt, könnt ihr es mit `cruft link TEMPLATE_REPOSITORY` mit der Vorlage verknüpfen, mit der es erstellt wurde, z.B.:

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

Ihr könnt dann den letzten Commit der Vorlage angeben, mit dem das Projekt aktualisiert wurde, oder die Vorgabe akzeptieren, den letzten Commit zu verwenden.

Diff anzeigen

Mit der Zeit kann sich euer Projekt stark von der eigentlichen Cookiecutter-Vorlage unterscheiden. `cruft diff` ermöglicht euch, schnell zu sehen, was sich in Ihrem lokalen Projekt im Vergleich zur Vorlage geändert hat.

12.6 Paket hochladen

Schließlich könnt ihr das Paket auf dem *Python Package Index* (*PyPI*) oder einem anderen Index, z.B. *GitLab Package Registry* oder *devpi*, bereitstellen.

Hierfür müsst ihr euch bei *Test PyPI* registrieren. *Test-PyPI* ist eine separate Instanz, die zum Testen und Experimentieren vorgesehen ist. Um dort ein Konto einzurichten, geht ihr auf https://test.pypi.org/account/register/. Weitere Informationen findet ihr unter Using TestPyPI.

Nun könnt ihr eine `~/.pypirc`-Datei erstellen:

```
[distutils]
index-servers=
    test

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

Siehe auch:

Wenn ihr die PyPI-Anmeldung automatisieren wollt, lest bitte Careful With That PyPI.

Nachdem ihr registriert seid, könnt ihr euer *Distribution Package* mit *twine* hochladen. Hierzu müsst ihr jedoch zunächst twine installieren mit:

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

Bemerkung: Führt diesen Befehl vor jedem Release aus um sicherzustellen, dass alle Release-Tools auf dem neuesten Stand sind.

Nun könnt ihr eure *Distribution Packages* erstellen mit:

```
$ cd /path/to/your/distribution_package
$ rm -rf build dist
$ pyproject-build .
```

Nach der Installation von Twine könnt ihr alle Archive unter `/dist` auf den Python Package Index hochladen mit:

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

-r, --repository
: Das Repository zum Hochladen des Pakets.

  In unserem Fall wird `test`-Abschnitt aus der `~/.pypirc`-Datei verwendet.

-s, --sign
: signiert die hochzuladenden Dateien mit GPG.

Ihr werdet nach eurem Passwort gefragt, mit dem ihr euch bei *Test PyPI* registriert habt. Anschließend solltet ihr eine ähnliche Ausgabe sehen:

```
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]
```

Bemerkung: Wenn ihr eine ähnliche Fehlermeldung erhaltet wie

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

müsst ihr einen eindeutigen Namen für euer Paket auswählen:

1. ändert das `name`-Argument in der `setup.py`-Datei
2. entfernt das `dist`-Verzeichnis
3. generiert die Archive neu

12.6.1 Überprüfen

Installation

Ihr könnt *pip* verwenden um euer Paket zu installieren und zu überprüfen, ob es funktioniert. Erstellt eine neue *virtuelle Umgebung* und installiert euer Paket von *Test PyPI*:

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

Bemerkung: Wenn ihr einen anderen Paketnamen verwendet habt, ersetzt ihn im obigen Befehl durch euren Paketnamen.

---

pip sollte das Paket von *Test PyPI* installieren und die Ausgabe sollte in etwa so aussehen:

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

Ihr könnt testen, ob euer Paket korrekt installiert wurde indem ihr das Modul importiert und auf die `name`-Eigenschaft referenziert, die zuvor in `__init__.py` eingegeben wurde:

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

Bemerkung: Die Pakete auf *Test-PyPI* werden nur temporär gespeichert. Wenn ihr ein Paket in den echten *Python Package Index* (*PyPI*) hochladen wollt, könnt ihr dies tun, indem ihr ein Konto auf *pypi.org* anlegt und die gleichen Anweisungen befolgt, jedoch `twine upload dist/*` verwendet.

README

Überprüft bitte auch, ob die `README.rst`-Datei auf der Test-PyPI-Seite korrekt angezeigt wird.

12.6.2 PyPI

Registriert euch nun beim *Python Package Index* (*PyPI*) und stellt sicher, dass die Zwei-Faktor-Authentifizierung aktiviert ist indem ihr die `~/.pypirc`-Datei ergänzt:

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

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

[pypi]
username = __token__
```

Mit dieser Konfiguration wird nicht mehr die Name/Passwort-Kombination beim Hochladen verwendet sondern ein Upload-Token.

Siehe auch:

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

Schließlich könnt ihr nun euer Paket auf PyPI veröffentlichen:

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

Bemerkung: Ihr könnt Releases nicht einfach ersetzen da ihr Pakete mit derselben Versionsnummer nicht erneut hochladen könnt.

Bemerkung: Entfernt nicht alte Versionen aus dem Python Package Index. Dies verursacht nur Arbeit für jene, die diese Version weiter verwenden wollen und dann auf alte Versionen auf GitHub ausweichen müssen. PyPI hat eine yank-Funktion, die ihr stattdessen nutzen könnt. Dies ignoriert eine bestimmte Version, wenn sie nicht explizit mit == oder === angegeben wurde.

Siehe auch:

- PyPI Release Checklist

12.6.3 GitHub Action

Ihr könnt auch eine GitHub-Aktion erstellen, die ein Paket erstellt und auf PyPI hochlädt. Eine solche `.github/workflows/pypi.yml`-Datei könnte folgendermaßen aussehen:

```yaml
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
    - name: Set up Python
      uses: actions/setup-python@v5
      with:
        python-version: '3.11'
        cache: pip
        cache-dependency-path: '**/pyproject.toml'
    - name: Install dependencies
      run: |
        python -m pip install -U pip
        python -m pip install -U setuptools build twine wheel
    - name: Build
      run: |
        python -m build
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
31      - name: Publish
32        env:
33          TWINE_PASSWORD: ${{ secrets.TWINE_PASSWORD }}
34          TWINE_USERNAME: ${{ secrets.TWINE_USERNAME }}
35        run: |
36          twine upload dist/*
```

Zeilen 3–5
Dies stellt sicher, dass der Arbeitsablauf jedes Mal ausgeführt wird, wenn ein neues GitHub-Release für das Repository erstellt wird.

Zeile 12
Der Job wartet auf das Bestehen des `test`-Jobs bevor er ausgeführt wird.

Siehe auch:

- GitHub Actions
- *cibuildwheel*

12.6.4 Trusted Publishers

Trusted Publishers ist ein alternatives Verfahren zum Veröffentlichen von Paketen auf dem *PyPI*. Sie basiert auf OpenID Connect und erfordert weder Passwort noch Token. Dazu sind lediglich die folgenden Schritte erforderlich:

1. Fügt einen *Trusted Publishers* auf PyPI hinzu

   Je nachdem, ob ihr ein neues Paket veröffentlichen oder ein bestehendes aktualisieren wollt, unterscheidet sich der Prozess geringfügig:

   - zum Aktualisieren eines bestehenden Pakets siehe Adding a trusted publisher to an existing PyPI project
   - zum veröffentlichen eines neuen Pakets gibt es ein spezielles Verfahren, *Pending Publisher* genannt; s.a. Creating a PyPI project with a trusted publisher

     Ihr könnt damit auch einen Paketnamen reservieren, bevor ihr die erste Version veröffentlicht. Damit könnt ihr sicherstellen, dass ihr das Paket auch unter dem gewünschten Namen veröffentlichen könnt.

     Hierfür müsst ihr in pypi.org/manage/account/publishing/ einen neuen *Pending Publisher* erstellen mit

     – Namen des PyPI-Projekts
     – GitHub-Repository Owner
     – Namen des Workflows, z.B. `publish.yml`
     – Name der Umgebung (optional), z.B. `release`

2. Erstellt eine Umgebung für die GitHub-Actions

   Wenn wir eine Umgebung auf *PyPI* angegeben haben, müssen wir diese nun auch erstellen. Das kann in *Settings* → *Environments* für das Repository geschehen. Der Name unserer Umgebung ist `release`.

3. Konfiguriert den Arbeitsablauf

   Hierfür erstellen wir nun die Datei `.github/workflows/publish.yml` in unserem Repository:

```
1  ...
2  jobs:
3    ...
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
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
```

Zeile 6
Dies wird benötigt, weil wir eine Umgebung in *PyPI* konfiguriert haben.

Zeilen 7–8
Sie sind erforderlich, damit die OpenID Connect-Token-Authentifizierung funktioniert.

Zeilen 19–20
Das Paket verwendet die Aktion github.com/pypa/gh-action-pypi-publish, um das Paket zu veröffentlichen.

12.7 GitLab Package Registry

Ihr könnt eure Verteilungspakete auch in der Paketregistrierung eures GitLab-Projekts veröffentlichen und sowohl mit *Pip* als auch mit *twine* nutzen.

Siehe auch:

PyPI packages in the Package Registry

12.7.1 Authentifizierung

Zur Authentifizierung an der GitLab Package Registry könnt ihr eine der folgenden Methoden verwenden:

- Ein *persönliches Zugriffstoken* mit dem Geltungsbereich `api`.
- Ein *Deploy-Token* mit den Geltungsbereichen `read_package_registry`, `write_package_registry` oder beiden.
- Ein *CI-Job-Token*.

... mit einem persönlichen Zugriffstoken

Um euch mit einem persönlichen Zugriffstoken zu authentifizieren, könnt ihr in der `~/.pypirc`-Datei z.B. folgendes hinzufügen:

```
[distutils]
index-servers=
    gitlab

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

... mit einem 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}
```

... mit einem 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/*
```

... für den Zugriff auf Pakete innerhalb einer Gruppe

Verwendet statt der *PROJECT_ID* die *GROUP_URL*.

12.7.2 Veröffentlichen des Verteilungspakets

Ihr könnt euer Paket mit Hilfe von *twine* veröffentlichen:

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

Bemerkung: Wenn ihr versucht, ein Paket zu veröffentlichen, das bereits mit demselben Namen und derselben Version existiert, erhaltet ihr den Fehler `400 Bad Request`; ihr müssen das vorhandene Paket dann zuerst löschen.

12.7.3 Installieren des Pakets

Ihr könnt die neueste Version eures Pakets installieren z.B. mit

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

... oder von der Gruppenebene aus mit

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

... oder in der `requirements.txt`-Datei mit

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

12.8 cibuildwheel

cibuildwheel vereinfacht die Erstellung von *Python Wheels* für die verschiedenen Plattformen und Python-Versionen durch Continuous Integration (CI) Workflows. Genauer gesagt baut es Manylinux-, macOS 10.9+- und Windows-Wheels für CPython und PyPy mit GitHub Actions, Azure Pipelines, Travis CI, AppVeyor, CircleCI, oder GitLab CI/CD.

Darüber hinaus bündelt es gemeinsam genutzte Bibliotheksabhängigkeiten unter Linux und macOS durch auditwheel und delocate.

Schließlich können die Tests auch gegen die Wheels laufen.

Siehe auch:

- Docs
- GitHub

12.8.1 GitHub Actions

Um Linux-, macOS- und Windows-Wheels erstellen zu können, erstellt eine `.github/workflows/build_wheels.yml`-Datei in eurem GitHub Repo:

```
name: Build

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

workflow_dispatch
: ermöglicht euch, in der grafischen Benutzeroberfläche auf eine Schaltfläche zu klicken, um einen Build auszulösen. Das ist perfekt zum manuellen Testen von Wheels vor einer Veröffentlichung eignet, da ihr sie einfach von *artifacts* herunterladen könnt.

Siehe auch:

- workflow_dispatch

release
: wird bei der Übertragung einer getaggten Version ausgeführt.

Siehe auch:

- release

Nun können die *Wheels* gebaut werden mit:

```
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
```

Dadurch wird der CI-Workflow mit den folgenden Standardeinstellungen ausgeführt:

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

Ihr könnt die Datei auch erweitern um die Wheels automatisch auf den *Python Package Index* (*PyPI*) hochzuladen. Hierfür solltet ihr jedoch zunächst noch eine *Source Distribution* erstellen, z.B. mit:

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    steps:
    - 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
```

Zudem muss dieser GitHub-Workflow in den PyPI-Einstellungen eures Projekts eingestellt werden:

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

Nun könnt ihr endlich die Artefakte beider Jobs auf den *PyPI* hochladen:

```
  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
```

Siehe auch:

- Workflow syntax for GitHub Actions

12.8.2 GitLab CI/CD

Um Linux-Wheels mit GitLab CI/CD zu bauen, erstellt eine `.gitlab-ci.yml`-Datei in eurem 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/
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    DOCKER_DRIVER: overlay2
    # 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
```

Siehe auch:

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

12.8.3 Optionen

`cibuildwheel` kann entweder über Umgebungsvariablen oder über eine Konfigurationsdatei wie `pyproject.toml` konfiguriert werden, z.B.:

```
[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"]
```

Siehe auch:

- cibuildwheel: Options

12.8.4 Beispiele

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

12.9 Binäre Erweiterungen

Eine der Funktionen des CPython-Interpreters besteht darin, dass neben der Ausführung von Python-Code auch eine reichhaltige C-API für die Verwendung durch andere Software verfügbar ist. Eine der häufigsten Anwendungen dieser C-API besteht darin, importierbare C-Erweiterungen zu erstellen, die Dinge ermöglichen, die im reinen Python-Code nur schwer zu erreichen sind.

12.9.1 Anwendungsfälle

Die typischen Anwendungsfälle für Binäre Erweiterungen lassen sich in drei Kategorien unterteilen:

Beschleunigungsmodule
: Diese Module sind eigenständig und werden nur erstellt, um schneller zu laufen als der entsprechende reine Python-Code. Im Idealfall haben die Accelerator-Module immer ein Python-Äquivalent, das als Fallback verwendet werden kann, wenn die beschleunigte Version auf einem bestimmten System nicht verfügbar ist.

  Die CPython-Standardbibliothek verwendet viele Accelerator-Module.

Wrapper-Module
: Diese Module werden erstellt, um vorhandene C-Interfaces in Python verfügbar zu machen. Sie können entweder die zugrunde liegenden C-Interfaces direkt verfügbar oder eine *Pythonic*-API bereitgestellt werden, die Features von Python verwendet, um die API einfacher zu verwenden.

  Die CPython-Standardbibliothek verwendet umfangreiche Wrapper-Module.

Systemzugriffe auf niedriger Ebene
: Diese Module werden erstellt, um auf Funktionen der CPython-Laufzeitumgebung, des Betriebssystems oder der zugrundeliegenden Hardware zuzugreifen. Durch plattformspezifischen Code können mit solchen Erweiterungsmodulen Dinge erreicht werden, die mit reinem Python-Code nicht möglich wären.

  Eine Reihe von CPython-Standard-Bibliotheksmodulen sind in C geschrieben, um auf Interpreter-Interna zuzugreifen, die nicht auf der Sprachebene verfügbar sind.

  Eine besonders bemerkenswerte Eigenschaft von C-Erweiterungen ist, dass sie, den Global Interpreter Lock (GIL) von CPython bei lang andauernden Operationen freigeben können, unabhängig davon, ob diese Operationen CPU- oder IO-gebunden sind.

Nicht alle Erweiterungsmodule passen genau in die oben genannten Kategorien. So umfassen z.B. die in NumPy enthaltenen Erweiterungsmodule alle drei Anwendungsfälle:

- Sie verschieben innere Schleifen aus Geschwindigkeitsgründen auf C,
- umschließen externe Bibliotheken in C, FORTRAN und anderen Sprachen und
- verwenden Systemschnittstellen auf niedriger Ebene für CPython und das zugrunde liegende Betriebssystem, um die gleichzeitige Ausführung von vektorisierten Operationen zu unterstützen und das Speicherlayout von erstellten Objekten genau zu steuern.

12.9.2 Nachteile

Früher war der Hauptnachteil bei der Verwendung von Beschleunigungsmodulen, dass dadurch die Distribution der Software erschwert wurde. Heute ist dieser Nachteil durch *wheel* kaum noch vorhanden. Einige Nachteile bleiben dennoch:

- Die Installation aus den Sourcen bleibt weiterhin kompliziert.
- Ggf. gibt es kein passendes *wheel* für den verwendeten Build des CPython-Interpreters oder alternativen Interpretern wie PyPy, IronPython oder Jython.
- Die Wartung und Pflege der Pakete ist aufwändiger da die Maintainer nicht nur mit Python sondern auch mit einer anderen Sprache und der CPython C-API vertraut sein müssen. Zudem erhöht sich die Komplexität, wenn neben dem Beschleunigungsmodul auch eine Python-Fallback-Implementierung bereitgestellt wird.
- Schließlich funktionieren häufig auch Importmechanismen, wie der direkte Import aus ZIP-Dateien, nicht für Extensions-Module.

12.9.3 Alternativen

... zu Beschleunigungsmodulen

Wenn Extensions-Module nur verwendet werden, um Code schneller auszuführen, sollten auch eine Reihe anderer Alternativen in Betracht gezogen werden:

- Sucht nach vorhandenen optimierten Alternativen. Die CPython-Standardbibliothek enthält eine Reihe optimierter Datenstrukturen und Algorithmen, insbesondere in den builtins und den Modulen `collections` und `itertools`.

  Gelegentlich bietet auch der *Python Package Index* (*PyPI*) zusätzliche Alternativen. Manchmal kann ein Modul eines Drittanbieters die Notwendigkeit vermeiden, ein eigenes Accelerator-Modul zu erstellen.
- Für lange laufende Anwendungen kann der JIT-kompilierte PyPy-Interpreter eine geeignete Alternative zum Standard-CPython sein. Die Hauptschwierigkeit bei der Übernahme von PyPy besteht typischerweise in der Abhängigkeit von anderen Beschleunigungsmodulen. Während PyPy die CPython C API emuliert, verursachen Module, die darauf angewiesen sind, Probleme für das PyPy JIT, und die Emulation legt oft Defekte in Beschleunigungsmodulen offen, die CPython toleriert. (häufig bei Reference Counting Errors).
- Cython ist ein ausgereifter statischer Compiler, der den meisten Python-Code zu C-Extensions-Modulen kompilieren kann. Die anfängliche Kompilierung bietet einige Geschwindigkeitssteigerungen (durch Umgehung der CPython-Interpreter-Ebene), und Cythons optionale statische Typisierungsfunktionen können zusätzliche Möglichkeiten für Geschwindigkeitssteigerungen bieten. Für Python-Programmierer bietet Cython eine niedrigere Eintrittshürde relativ zu anderen Sprachen wie C oder C ++).

  Die Verwendung von Cython hat jedoch den Nachteil, die Komplexität der Verteilung der resultierenden Anwendung zu erhöhen.
- Numba ist ein neueres Tool, das die LLVM Compiler-Infrastruktur nutzt, um während der Laufzeit selektiv Teile einer Python-Anwendung auf nativen Maschinencode zu kompilieren. Es erfordert, dass LLVM auf dem System verfügbar ist, auf dem der Code ausgeführt wird. Es kann, insbesondere bei vektorisierbaren Vorgängen zu erheblichen Geschwindigkeitssteigerungen führen.

... zu Wrapper-Modulen

Die C-ABI (Application Binary Interface) ist ein Standard für die gemeinsame Nutzung von Funktionen zwischen mehreren Anwendungen. Eine der Stärken der CPython C-API (Application Programming Interface) ist es, dass Python-Benutzer diese Funktionalität nutzen können. Das manuelle Wrapping von Modulen ist jedoch sehr mühsam, so dass eine Reihe anderer Alternativen in Betracht gezogen werden sollten.

Die unten beschriebenen Ansätze vereinfachen nicht die Distribution, aber sie können den Wartungsaufwand im Vergleich zu Wrapper-Modulen deutlich reduzieren.

- Cython eignet sich nicht nur zum Erstellen von Accelerator-Modulen, sondern auch zum Erstellen von Wrapper-Modulen. Da das Wrapping der API immer noch von Hand erfolgen muss, ist es keine gute Wahl beim Wrapping großer APIs.
- cffi ist das Projekt einiger Personen aus dem PyPy-Entwicklungsteam, um C-Module einfacher für Python-Anwendungen verfügbar zu machen. Es macht das Wrapping eines C-Moduls basierend auf seinen Header-Dateien relativ einfach, auch wenn man sich mit C selbst nicht auskennt.

  Einer der Hauptvorteile von cffi besteht darin, dass es mit dem PyPy-JIT kompatibel ist, sodass CFFI-Wrapper-Module vollständig von den PyPy-Tracing-JIT-Optimierungen partizipieren können.
- SWIG ist ein Wrapper Interface Generator, der eine Vielzahl von Programmiersprachen, einschließlich Python, mit C- und C++-Code verbindet.
- Das `ctypes`-Modul der Standardbibliothek ist zwar nützlich um Zugriff auf C-Schnittstellen zu erhalten, wenn die Header-Informationen jedoch nicht verfügbar sind, leidet es jedoch daran, dass es nur auf der C ABI-Ebene arbeitet und somit keine automatische Konsistenzprüfung zwischen der exportierten Schnittstelle und dem Python-Code macht. Im Gegensatz dazu können die obigen Alternativen alle auf der C-API arbeiten und C-Header-Dateien verwenden, um die Konsistenz zu gewährleisten.
- pythoncapi_compat kann verwendet werden, um eine C-Erweiterung zu schreiben, die mehrere Python-Versionen mit einer einzigen Codebasis unterstützt. Es besteht aus der Header-Datei `pythoncapi_compat.h` und dem Skript `upgrade_pythoncapi.py`.

... für den Systemzugriff auf niedriger Ebene

Für Anwendungen, die Low Level System Access benötigen, ist ein Beschleunigungsmodul oft ist der beste Weg. Dies gilt insbesondere für den Low Level Access auf die CPython-Runtime, da einige Operationen (wie das Freigeben des Global Interpreter Lock (GIL) nicht zulässig sind, wenn der Interpreter den Code selbst ausführt, gerade auch wenn Module wie `ctypes` oder `cffi` verwendet werden, um Zugriff auf das relevanten C-API-Interfaces zu erhalten.

In Fällen, in denen das Erweiterungsmodul das zugrunde liegende Betriebssystem oder die Hardware (statt der CPython-Runtime) manipuliert, ist es manchmal besser, eine normale C-Bibliothek (oder eine Bibliothek in einer anderen Programmiersprache wie C ++ oder Rust) zu schreiben, die eine C-kompatible ABI), bereitstellt und anschließend eine der oben beschriebenen Wrapping-Techniken zu verwenden um das Interface als importierbares Python-Modul verfügbar zu machen.

12.9.4 Implementierung

Wir wollen nun unser `dataprep`-Paket erweitern und einigen C-Code integrieren. Hierfür verwenden wir Cython, um den Python-Code aus `dataprep/src/dataprep/cymean.pyx` in optimierten C-Code während des Build-Prozesses zu übersetzen. Cython-Dateien haben den Suffix `pyx` und können sowohl Python- also auch C-Code enthalten.

Als Build-Backend können wir jedoch aktuell noch nicht `hatchling.build` verwenden, sondern greifen auf eine aktuelle Version der *setuptools* zurück:

```
[build-system]
requires = ["Cython", "setuptools>=61.0"]
build-backend = "setuptools.build_meta"
```

Bemerkung: Mit extensionlib gibt es ein Toolkit für Extensions-Module, das aktuell jedoch noch kein `hatchling`-Plugin enthält.

Bemerkung: Alternativ könntet ihr auch *Meson* oder *scikit-build* verwenden:

```
[build-system]
requires = ["meson-python"]
build-backend = "mesonpy"
```

```
[build-system]
requires = ["scikit-build-core"]
build-backend = "scikit_build_core.build"
```

Da Cython selbst ein Python-Paket ist, kann es einfach in der `dataprep/pyproject.toml`-Datei in die Liste der Abhängigkeiten aufgenommen werden:

```
dependencies = [
    "Cython",
    "pandas",
]
```

Die Setuptools nutzen `dataprep/setup.py`, um auch Nicht-Python-Dateien in ein Paket aufzunehmen.

```
from Cython.Build import cythonize
from setuptools import find_packages, setup

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

Nun könnt ihr den Build-Prozess mit dem Befehl `pyproject-build` ausführen und überprüfen, ob die Cython-Datei auch wie erwartet im Paket landet:

```
$ 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.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
[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
```

Schließlich können wir unser Paket überprüfen mit `check-wheel-contents`:

```
$ check-wheel-contents dataprep/dist/*.whl
dataprep/dist/dataprep-0.1.0-cp39-cp39-macosx_10_9_x86_64.whl: OK
```

Alternativ könnt ihr auch unser `dataprep`-Paket installieren und `mean` verwenden:

```
$ 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)
500.296087
```

Dabei wurde mit der `random.randint`-Funktion eine Liste von einer Million Zufallszahlen mit Werten zwischen 1 und 1000 erstellt.

Siehe auch:

Der CPython Extending and Embedding guide enthält eine Einführung in das Schreiben eigener Extension-Module in C: Extending Python with C or C++. Beachtet jedoch bitte, dass diese Einführung nur die grundlegenden Tools zum Erstellen von Erweiterungen beshreibt, die im Rahmen von CPython bereitgestellt werden. Third-Party-Tools wie Cython, cffi, SWIG und Numba bieten sowohl einfachere als auch ausgeklügeltere Ansätze zum Erstellen von C- und C ++- Erweiterungen für Python.

Python Packaging User Guide: Binary Extensions behandelt nicht nur verschiedene verfügbare Tools, die die Erstellung von Beschleunigungsmodulen vereinfachen, sondern erläutert auch die verschiedenen Gründe, warum das Erstellen eines Extension Module wünschenswert sein könnte.

12.9.5 Erstellen von Beschleunigungsmodulen

Beschleunigungsmodule für Windows

Bevor ihr ein Beschleunigungsmodul erstellen könnt, müsst ihr sicherstellen, dass ihr einen geeigneten Compiler zur Verfügung habt. Unter Windows wird Visual C zum Erstellen des offiziellen CPython-Interpreters verwendet und er sollte auch zum Erstellen kompatibler Beschleunigungsmodule verwendet werden:

Für Python 3.5 installiert Visual Studio Code mit Python Extension

Das Erstellen mit dem empfohlenen Compiler unter Windows stellt sicher, dass eine kompatible C-Bibliothek im gesamten Python-Prozess verwendet wird.

Beschleunigungsmodule für Linux

Linux-Binaries müssen eine ausreichend alte glibc verwenden, um mit älteren Distributionen kompatibel zu sein. Distrowatch bereitet in tabellarischer Form auf, welche Versionen der Distributionen welche Bibliothek liefern:

- Red Hat Enterprise Linux
- Debian
- Ubuntu
- ...

Das PYPA/Manylinux-Projekt erleichtert die Distribution von Beschleunigungsmodulen als *Wheels* für die meisten Linux-Plattformen. Hieraus ging auch **PEP 513** hervor, das die `manylinux1_x86_64`- und `manylinux1_i686`-Plattform-Tags definiert.

Beschleunigungsmodule für macOS

Die Binärkompatibilität auf macOS wird durch das Zielsystem für die minimale Implementierung bestimmt, z. B. *10.9* , das in der Umgebungsvariable `MACOSX_DEPLOYMENT_TARGET` definiert wird. Beim Erstellen mit setuptools/distutils wird das Deployment-Ziel mit dem Flag `--plat-name` angegeben, z.B. `macosx-10.9-x86_64`. Weitere Informationen zu Deployment-Zielen für Mac OS Python-Distributionen findet ihr im MacPython Spinning Wheels-Wiki.

12.9.6 Deployment von Beschleunigungsmodulen

Im Folgenden soll das Deployment auf dem *Python Package Index* (*PyPI*) oder einem anderen Index beschrieben werden.

Bemerkung: Bei Deployments auf Linux-Distributionen sollte beachtet werden, dass diese Anforderungen an das spezifische Build-System stellen. Daher sollten neben *Wheels* immer auch *Source Distributions (sdist)* bereitgestellt werden.

Siehe auch:

- Deploying Python applications
- Supporting Windows using Appveyor

12.10 Glossar

build
: `build` ist ein **PEP 517**-kompatibler Python-Paket-Builder. Er bietet eine CLI zum Erstellen von Paketen sowie eine Python-API.

  Docs | GitHub | PyPI

Built Distribution
bdist
: Eine Struktur aus Dateien und Metadaten, die bei der Installation nur an den richtigen Speicherort auf dem Zielsystem verschoben werden müssen. *wheel* ist ein solches Format, nicht jedoch *distutil's Source Distribution*, die einen Build-Schritt erfordern.

cibuildwheel
: *cibuildwheel* ist ein Python-Paket, das *wheels* für alle gängigen Plattformen und Python-Versionen auf den meisten CI-Systemen erstellt.

  Docs | GitHub | PyPI

  Siehe auch:

  multibuild

conda
: Paketmanagement-Tool für die Anaconda-Distribution von Continuum Analytics. Sie ist speziell auf die wissenschaftliche Gemeinschaft ausgerichtet, insbesondere auf Windows, wo die Installation von binären Erweiterungen oft schwierig ist.

  Conda installiert keine Pakete von *PyPI* und kann nur von den offiziellen Continuum-Repositories oder von anaconda.org oder lokalen (z.B. Intranet-) Paketservern installieren. Beachtet jedoch, dass *pip* in conda installiert werden und Seite an Seite arbeiten kann, um Distributionen von *PyPI* zu verwalten.

  Siehe auch:

  - Conda: Myths and Misconceptions
  - Conda build variants

devpi
: devpi ist ein leistungsstarker *PyPI*-kompatibler Server und ein PyPI-Proxy-Cache mit einem Befehlszeilenwerkzeug um Paketierungs-, Test- und Veröffentlichungsaktivitäten zu ermöglichen.

  Docs | GitHub | PyPI

Distribution Package
: Eine versionierte Archivdatei, die Python-*Pakete*, -*Module* und andere Ressourcendateien enthält, die zum Verteilen eines *Releases* verwendet werden.

distutils
: Paket der Python-Standardbibliothek, das Unterstützung für das Bootstrapping von *pip* in eine bestehende Python-Installation oder *virtuelle Umgebung* bietet.

  Docs | GitHub

Egg
: Ein *Built Distribution*-Format, das von *Setuptools* eingeführt wurde und nun durch *wheel* ersetzt wird. Weitere Informationen findet ihr unter The Internal Structure of Python Eggs und Python Eggs.

enscons
: enscons ist ein Python-Paketierungswerkzeug, das auf SCons basiert. Es erstellt *pip*-kompatible *Source Distributions* und *wheels* ohne Verwendung von *distutils* oder *setuptools*, einschließlich Distributionen mit C-Erweiterungen. enscons hat eine andere Architektur und Philosophie als *distutils*, da es Python-Paketierung zu einem allgemeinen Build-System hinzufügt. enscons kann euch helfen, *sdists* und *wheels* zu bauen.

  GitHub | PyPI

Flit
: Flit bietet eine einfache Möglichkeit, reine Python-Pakete und -Module zu erstellen und auf den *Python Package Index* hochzuladen. Flit kann eine Konfigurationsdatei generieren, um schnell ein Projekt einzurichten, eine *Source Distribution* und ein *wheel* zu erstellen und sie zu PyPI hochzuladen.

  Flit verwendet *pyproject.toml*, um ein Projekt zu konfigurieren. Flit ist nicht auf Werkzeuge wie *setuptools* angewiesen, um Distributionen zu erstellen, oder auf *twine*, um sie auf *PyPI* hochzuladen.

  Docs | GitHub | PyPI

Hatch
: Hatch ist ein Kommandozeilenwerkzeug, das ihr zum Konfigurieren und Versionieren von Paketen, zum Spezifizieren von Abhängigkeitengenutzt werden kann. Das Plugin-System ermöglicht die einfache Erweiterung der Funktionalitäten.

  Docs | GitHub | PyPI

hatchling
: Build-Backend von *Hatch*, das auch zum Veröffentlichen auf dem *Python Package Index* genutzt werden kann.

Import Package
: Ein Python-Modul, das andere Module oder rekursiv andere Pakete enthalten kann.

maturin
: Vormals pyo3-pack, ist ein **PEP 621**-kompatibles Build-Tool für *binäre Erweiterungen* in Rust.

meson-python
: Build-Backend, das das Meson-Build-System verwendet. Es unterstützt eine Vielzahl von Sprachen, einschließlich C, und ist in der Lage, die Anforderungen der meisten komplexen Build-Konfigurationen zu erfüllen.

  Docs | GitHub | PyPI

Modul
: Die Grundeinheit der Wiederverwendbarkeit von Code in Python, die in einem von zwei Typen existiert:

  Pure Module
  : Ein Modul, das in Python geschrieben wurde und in einer einzigen `.py`-Datei enthalten ist (und möglicherweise zugehörigen `.pyc`- und/oder `.pyo`-Dateien).

  Extension Module
  : In der Regel in eine einzelne dynamisch ladbare vorkompilierte Datei, z. B. einer gemeinsamen Objektdatei (`.so`).

multibuild
: `multibuild` ist ein Satz von CI-Skripten zum Erstellen und Testen von Python-*wheels* für Linux, macOS und Windows.

  Siehe auch:

  cibuildwheel

pdm
: Python-Paketmanager mit **PEP 582**-Unterstützung. Er installiert und verwaltet Pakete ohne dass eine *virtuelle Umgebung* erstellt werden muss. Er verwendet auch *pyproject.toml*, um Projekt-Metadaten zu speichern, wie in **PEP 621** definiert.

  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
: Beliebtes Werkzeug für die Installation von Python-Paketen, das in neuen Versionen von Python enthalten ist.

  Es bietet die wesentlichen Kernfunktionen zum Suchen, Herunterladen und Installieren von Paketen aus dem *Python Package Index* und andere Python-Paketverzeichnissen und kann über eine Befehlszeilenschnittstelle (CLI) in eine Vielzahl von Entwicklungsabläufen eingebunden werden.

  Docs | GitHub | PyPI

pip-tools
: Reihe von Werkzeugen, die eure Builds deterministisch halten und dennoch mit neuen Versionen eurer Abhängigkeiten auf dem Laufenden halten können.

  Docs | GitHub | PyPI

Pipenv
: Pipenv bündelt *Pipfile*, *pip* und *virtualenv* in einer einzigen Toolchain. Es kann die `requirements.txt` automatisch importieren und mithilfe von safety die Umgebung auch auf CVEs prüfen. Schließlich erleichtert es auch die Deinstallation von Paketen und deren Abhängigkeiten.

  Docs | GitHub | PyPI

Pipfile
Pipfile.lock
: `Pipfile` und `Pipfile.lock` sind eine übergeordnete, anwendungsorientierte Alternative zu *pip*'s `requirements.txt`-Datei. Die **PEP 508 Environment Markers** werden ebenfalls unterstützt.

  Docs | GitHub

pipx
: pipx untertüzt euch, Abhängigkeitskonflikte mit anderen auf dem System installierten Paketen zu vermeiden.

  Docs | GitHub | PyPI

piwheels
: Website und zugrundeliegende Software, die *Source Distribution*-Pakete von *PyPI* holt und sie in binäre *wheels* kompiliert, die für die Installation auf Raspberry Pis optimiert sind.

  Home | Docs | GitHub

poetry
: Eine All-in-One-Lösung für reine Python-Projekte. Es ersetzt *setuptools*, *venv*/*pipenv*, *pip*, *wheel* und *twine*. Sie macht jedoch einige schlechte Standardannahmen für Bibliotheken und die *pyproject.toml*-Konfiguration ist nicht standardkonform.

  Docs | GitHub | PyPI

pybind11
: Dies ist *setuptools*, aber mit einer C++-Erweiterung und von *cibuildwheel* generierten *wheels*.

  Docs | GitHub | PyPI

pypi.org
: pypi.org ist der Domainname für den *Python Package Index* (*PyPI*). Er löste 2017 den alten Index-Domain-Namen `pypi.python.org` ab. Er wird von *warehouse* unterstützt.

pyproject.toml
: Werkzeugunabhängige Datei zur Spezifikation von Projekten, die in **PEP 518** definiert ist.

  Docs

  Siehe auch:

  - *pyproject.toml*

Python Package Index
PyPI
: *pypi.org* ist der Standard-Paket-Index für die Python-Community. Alle Python-Entwickler können ihre Distributionen nutzen und verteilen.

Python Packaging Authority
PyPA
: Die Python Packaging Authority ist eine Arbeitsgruppe, die mehrere Softwareprojekte für die Paketierung, Ver-

teilung und Installation von Python-Bibliotheken verwaltet. Die in PyPA Goals genannten Ziele sind jedoch noch während der Diskussionen um **PEP 516**, **PEP 517** und **PEP 518** entstanden, die mit dem *pyproject.toml*-basierten Build-System konkurrierende Workflows erlaubten, die nicht interoperabel sein müssen.

readme_renderer
: `readme_renderer` ist eine Bibliothek, die verwendet wird, um Dokumentation aus Auszeichnungssprachen wie Markdown oder reStructuredText in HTML zu rendern. Ihr könnt sie verwenden, um zu prüfen, ob eure Paketbeschreibungen auf *PyPI* korrekt angezeigt werden.

GitHub | PyPI

Release
: Der Snapshot eines Projekts zu einem bestimmten Zeitpunkt, gekennzeichnet durch eine Versionskennung.

Eine Veröffentlichung kann mehrere *Built Distributions* zur Folge haben.

scikit-build
: Build-System-Generator für `C`-, `C++`-, `Fortran`- und `Cython`-Erweiterungen, der *setuptools*, *wheel* und *pip* integriert. Er verwendet intern `CMake`, um eine bessere Unterstützung für zusätzliche Compiler, Build-Systeme, Cross-Compilation und das Auffinden von Abhängigkeiten und deren zugehörigen Build-Anforderungen zu bieten. Um die Erstellung großer Projekte zu beschleunigen und zu parallelisieren, kann zusätzlich Ninja installiert werden.

Docs | GitHub | PyPI

setuptools
: setuptools sind das klassische Build-System, das sehr leistungsfähig ist, aber mit steiler Lernkurve und hohem Konfigurationsaufwand. Ab Version 61.0.0 unterstützen die setuptools auch *pyproject.toml*-Dateien.

Docs | GitHub | PyPI

Siehe auch:

Packaging and distributing projects

shiv
: Kommandozeilenprogramm zur Erstellung von Python-Zip-Apps, wie sie in **PEP 441** beschrieben sind, aber zusätzlich mit allen Abhängigkeiten.

Docs | GitHub | PyPI

Source Distribution
sdist
: Ein Verteilungsformat (das normalerweise mithilfe von `python setup.py sdist` generiert wird).

Es stellt Metadaten und die wesentlichen Quelldateien bereit, die für die Installation mit einem Tool wie *Pip* oder zum Generieren von *Built Distributions* benötigt werden.

Spack
: Flexibler Paketmanager, der mehrere Versionen, Konfigurationen, Plattformen und Compiler unterstützt. Beliebig viele Versionen von Paketen können auf demselben System koexistieren. Spack wurde für die schnelle Erstellung von wissenschaftlichen Hochleistungsanwendungen auf Clustern und Supercomputern entwickelt.

Docs | GitHub

Siehe auch:

- Spack

trove-classifiers
: trove-classifiers sind zum einen Klassifikatoren, die im *Python Package Index* verwendet werden, um Projekte systematisch zu beschreiben und besser auffindbar zu machen. Zum anderen sind sie ein Paket, das eine Liste gültiger und veralteter Klassifikatoren enthält, das zur Überprüfung verwendet werden kann.

Docs | GitHub | PyPI

twine
: Kommandozeilenprogramm, das Programmdateien und Metadaten an eine Web-API übergibt. Damit lassen sich Python-Pakete auf den *Python Package Index* hochladen.

  Docs | GitHub | PyPI

venv
: Paket, das ab Python 3.3 in der Python-Standardbibliothek ist und zur Erstellung *virtueller Umgebungen* gedacht ist.

  Docs | GitHub

virtualenv
: Werkzeug, das die Befehlszeilen-Umgebungsvariable `path` verwendet, um isolierte *virtuelle Python-Umgebungen* zu erstellen, ähnlich wie *venv*. Es bietet jedoch zusätzliche Funktionalität für die Konfiguration, Wartung, Duplizierung und Fehlerbehebung.

  Ab Version 20.22.0 unterstützt virtualenv nicht mehr die Python-Versionen 2.7, 3.5 und 3.6.

Virtuelle Umgebung
: Eine isolierte Python-Umgebung, die die Installation von Paketen für eine bestimmte Anwendung ermöglicht, anstatt sie systemweit zu installieren.

  Siehe auch:

  - *Virtuelle Umgebungen*
  - Creating Virtual Environments

Warehouse
: Die aktuelle Codebasis, die den *Python Package Index* (*PyPI*) antreibt. Sie wird auf *pypi.org* gehostet.

  Docs | GitHub

wheel
: Distributionsformat, das mit **PEP 427** eingeführt wurde. Es soll das *Egg*-Format ersetzen und wird von aktuellen *pip*-Installationen unterstützt.

  C-Erweiterungen können als plattformspezifische wheels für Windows, macOS und Linux auf dem *PyPI* bereitgestellt werden. Dies hat für euch den Vorteil, dass ihr bei der Installation des Pakets dieses nicht kompilieren zu müssen.

  Home | Docs | **PEP 427** | GitHub | PyPI

  Siehe auch:

  - *wheels*

whey
: Einfacher Python-*wheel*-Builder mit Automatisierungsoptionen für *trove-classifiers*.

KAPITEL 13

Objektorientierung

Python bietet volle Unterstützung für Objektorientierte Programmierung (OOP (Objektorientierte Programmierung)).

13.1 Klassen

Eine Klasse in Python ist eigentlich ein Datentyp. Alle in Python eingebauten Datentypen sind Klassen, und Python stellt euch leistungsfähige Werkzeuge bereit, um jeden Aspekt des Verhaltens einer Klasse zu manipulieren. Ihr könnt eine Klasse mit der `class`-Anweisung definieren:

```
>>> class MyClass:
...     STATEMENTS
...
```

`MyClass`
: Klassenbezeichner werden üblicherweise in Großbuchstaben geschrieben, D.H. der erste Buchstabe jedes Wortes wird großgeschrieben, um die Bezeichner hervorzuheben.

`STATEMENTS`
: ist eine Liste von Python-Anweisungen – in der Regel Variablenzuweisungen und Funktionsdefinitionen. Es sind jedoch keine Zuweisungen oder Funktionsdefinitionen erforderlich, es kann auch nur eine einzige `pass`-Anweisung sein.

Nachdem ihr die Klasse definiert habt, könnt ihr ein neues Objekt des Klassentyps (eine Instanz der Klasse) erstellen, indem ihr den Klassennamen als Funktion aufruft:

```
>>> class Square:
...     length = 1
...
>>> my_square = Square()
```

Klasseninstanzen können als Strukturen oder Datensätze verwendet werden. Im Gegensatz zu C-Strukturen oder Java-Klassen müssen die Datenfelder einer Instanz jedoch nicht im Voraus deklariert werden. Das folgende kurze Beispiel

definiert eine Klasse namens `Square`, erstellt eine `Square`-Instanz, weist der Kantenlänge einen Wert zu und verwendet diesen Wert dann zur Berechnung des Umfangs:

```
>>> my_square.length = 3
>>> print(f"Der Umfang des Quadrats ist {4 * my_square.length}.")
Der Umfang des Quadrats ist 12.
```

Zeile 1
: Wie in Java und vielen anderen Sprachen werden die Felder einer Instanz mit Hilfe der Punktnotation angesprochen.

Ihr könnt Felder einer Instanz automatisch initialisieren, indem ihr eine `__init__`-Initialisierungsmethode in die Klasse aufnehmt. Diese Funktion wird jedes Mal ausgeführt, wenn eine Instanz der Klasse mit dieser neuen Instanz als erstes Argument `self` erstellt wird. Anders als in Java und C++ können Python-Klassen auch nur eine `__init__`-Methode haben. Im folgenden Beispiel werden standardmäßig Quadrate mit einer Kantenlänge von 1 erzeugt:

```
1 >>> class Square:
2 ...     def __init__(self):
3 ...         self.length = 1
4 ...
5 >>> my_square = Square()
6 >>> print(f"Der Umfang des Quadrats ist {4 * my_square.length}.")
7 Der Umfang des Quadrats ist 4.
```

Zeile 2
: Der Konvention nach ist `self` immer der Name des ersten Arguments von `__init__`. `self` wird auf die neu erstellte `Square`-Instanz gesetzt, wenn `__init__` ausgeführt wird.

Zeile 5
: Als nächstes erstellt ihr ein `Square`-Instanzobjekt.

Zeile 6
: Diese Zeile nutzt die Tatsache, dass das `length`-Feld bereits initialisiert ist.

Ihr könnt das `length`-Feld auch überschreiben, so dass die letzte Zeile ein anderes Ergebnis ausgibt als die vorherige `print`-Anweisung:

```
>>> my_square.length = 3
>>> print(f"Der Umfang des Quadrats ist {4 * my_square.length}.")
Der Umfang des Quadrats ist 12.
```

13.2 Variablen

13.2.1 Instanzvariablen

Im vorigen Beispiel ist `length` eine Instanzvariable von `Square`-Instanzen, D.H., jede Instanz der Klasse `Square` hat ihre eigene Kopie von `length`, und der in dieser Kopie gespeicherte Wert kann sich von den Werten unterscheiden, die in der `length`-Variable in anderen Instanzen gespeichert sind. In Python könnt ihr Instanzvariablen nach Bedarf erstellen, indem ihr sie dem Feld einer Klasseninstanz zuweist. Wenn die Variable noch nicht existiert, wird sie automatisch erstellt.

Alle Verwendungen von Instanzvariablen, sowohl die Zuweisung als auch der Zugriff, erfordern die explizite Erwähnung der enthaltenen Instanz, D.H. `instance.variable`. Ein Verweis auf eine Variable an sich ist kein Verweis auf eine Instanzvariable, sondern auf eine lokale Variable in der ausführenden Methode. Dies ist ein Unterschied zu C++ und Java, wo Instanzvariablen auf die gleiche Weise referenziert werden wie lokale Funktionsvariablen der Methode.

Python schreibt hier die explizite Erwähnung der enthaltenen Instanz vor, und dies ermöglicht eine klare Unterscheidung zwischen Instanzvariablen und lokalen Funktionsvariablen.

13.2.2 Klassenvariablen

Eine Klassenvariable ist eine Variable, die mit einer Klasse verbunden ist, nicht mit einer Instanz einer Klasse, und auf die alle Instanzen der Klasse zugreifen können. Eine Klassenvariable kann verwendet werden, um einige Informationen auf Klassenebene zu speichern, z.B. wie viele Instanzen der Klasse zu einem bestimmten Zeitpunkt erstellt wurden. Python stellt Klassenvariablen zur Verfügung, obwohl deren Verwendung etwas mehr Aufwand erfordert als in den meisten anderen Sprachen. Außerdem müsst ihr auf eine Wechselwirkung zwischen Klassen- und Instanzvariablen achten.

Eine Klassenvariable wird durch eine Zuweisung in der Klasse, jedoch außerhalb der `__init__`-Funktion, erzeugt. Nachdem sie erstellt wurde, kann sie von allen Instanzen der Klasse gesehen werden. ihr könnt eine Klassenvariable verwenden, um einen Wert für `pi` für alle Instanzen der Klasse `Circle` zugänglich zu machen:

```
>>> class Circle:
...     pi = 3.14159
...     def __init__(self, diameter):
...         self.diameter = diameter
...     def circumference(self):
...         return self.diameter * Circle.pi
...
```

Wenn ihr diese Definition eingegeben habt, könnt ihr `pi` abfragen mit:

```
>>> Circle.pi
3.14159
```

Bemerkung: Die Klassenvariable ist mit der Klasse, die sie definiert, verknüpft und in ihr enthalten. Ihr greift in diesem Beispiel auf `Circle.pi` zu, bevor irgendwelche `Circle`-Instanzen erstellt wurden. Es ist offensichtlich, dass `Circle.pi` unabhängig von bestimmten Instanzen der Klasse `Circle` existiert.

Ihr könnt auch von einer Methode einer Klasse aus über den Klassennamen auf eine Klassenvariable zugreifen. Ihr tut dies in der Definition von `Circle.circumference`, wo die Funktion `circumference` einen speziellen Verweis auf `Circle.pi` enthält:

```
>>> c = Circle(3)
>>> c.circumference()
9.424769999999999
```

Unschön ist jedoch, dass der Klassenname `Circle` in der Methode `circumference` verwendet wird, um die Klassenvariable `pi` anzusprechen. Ihr könnt dies vermeiden, indem ihr das spezielle `__class__`-Attribut verwendet, das für alle Python-Klasseninstanzen verfügbar ist. Dieses Attribut gibt die Klasse zurück, zu der die Instanz gehört, z.B.:

```
>>> Circle
<class '__main__.Circle'>
>>> c.__class__
<class '__main__.Circle'>
```

Die Klasse `Circle` wird intern durch eine abstrakte Datenstruktur repräsentiert, und diese Datenstruktur ist genau das, was durch das `__class__`-Attribut von `c`, einer Instanz der Klasse `Circle`, erhalten wird. In diesem Beispiel könnt ihr den Wert von `Circle.pi` von `c` abrufen, ohne sich explizit auf den Namen der Klasse `Circle` zu beziehen:

```
>>> c.__class__.pi
3.14159
```

Ihr könnt diesen Code intern in der Methode `circumference` verwenden, um den expliziten Verweis auf die Klasse `Circle` loszuwerden; ersetzt `Circle.pi` durch `self.__class__.pi`.

Es gibt eine kleine Merkwürdigkeit bei Klassenvariablen, die euch verwirren könnte, wenn ihr euch dessen nicht bewusst seid.

Warnung: Wenn Python eine Instanzvariable sucht und keine Instanzvariable mit diesem Namen findet, wird der Wert in einer Klassenvariablen mit demselben Namen gesucht und zurückzugeben. Nur wenn keine passende Klassenvariable gefunden werden kann, gibt Python einen Fehler aus. Damit können zwar effizient Standardwerte für Instanzvariablen implementiert werden; dies führt jedoch auch leicht dazu, versehentlich auf eine Instanzvariable statt auf eine Klassenvariable zu verweisen, ohne dass ein Fehler gemeldet wird.

Zunächst könnt ihr euch auf die Variable `c.pi` beziehen, obwohl `c` keine zugehörige Instanzvariable namens `pi` hat. Python versucht zunächst, eine solche Instanzvariable zu finden und erst, wenn es keine Instanzvariable finden kann, wird eine Klassenvariable `pi` in `Circle` gesucht:

```
>>> c1 = Circle(1)
>>> c1.pi
3.14159
```

Wenn ihr nun feststellt, dass eure Angabe für `pi` zu früh gerundet wurde und ihr sie durch eine präzisere Angabe ersetzen wollt, könntet ihr geneigt sein, dies folgendermaßen zu ändern:

```
>>> c1.pi = 3.141592653589793
>>> c1.pi
3.141592653589793
```

Ihr habt jetzt jedoch lediglich `c1` eine neue Instanzvariable `pi` hinzugefügt. Die Klassenvariable `Circle.pi` und alle anderen daraus abgeleiteten Instanzen haben weiterhin nur fünf Nachkommastellen:

```
>>> Circle.pi
3.14159
>>> c2 = Circle(2)
>>> c1.pi
3.14159
```

13.3 Methoden

Eine Methode ist eine Funktion, die mit einer bestimmten Klasse verbunden ist. Ihr habt bereits die spezielle `__init__`-Methode kennengelernt, die bei einer neuen Instanz aufgerufen wird, wenn diese erstellt wird. Im folgenden Beispiel definiert ihr eine weitere Methode, `circumference`, für die Klasse `Square`; diese Methode kann verwendet werden, um den Umfang für eine beliebige `Square`-Instanz zu berechnen und zurückzugeben. Wie die meisten benutzerdefinierten Methoden wird `circumference` mit einer Syntax aufgerufen, die dem Zugriff auf Instanzvariablen ähnelt:

```
>>> class Square:
...     def __init__(self):
...         self.length = 1
...     def circumference(self):
...         return 4 * self.length
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
...
>>> s = Square()
>>> s.length = 5
>>> print(s.circumference())
20
```

Die Syntax für Methodenaufrufe besteht aus einer Instanz, gefolgt von einem Punkt, gefolgt von der Methode, die auf der Instanz aufgerufen werden soll. Wenn eine Methode auf diese Weise aufgerufen wird, handelt es sich um einen gebundenen Methodenaufruf. Eine Methode kann jedoch auch als ungebundene Methode aufgerufen werden, indem über ihre enthaltende Klasse auf sie zugegriffen wird. Diese Praxis ist weniger praktisch und wird fast nie angewandt, da das erste Argument einer Methode, die auf diese Weise aufgerufen wird, eine Instanz der Klasse sein muss, in der die Methode definiert ist, und weniger klar ist:

```
>>> print(Square.circumference(s))
20
```

Wie `__init__` wird auch die `circumference`-Methode als Funktion innerhalb der Klasse definiert. Das erste Argument jeder Methode ist die Instanz, von der oder auf der sie aufgerufen wurde, konventionsgemäß `self` genannt. In vielen Sprachen wird die Instanz `this` genannt und nie explizit übergeben.

Methoden können mit Argumenten aufgerufen werden, wenn die Methodendefinitionen diese Argumente akzeptieren. Diese Version von `Square` fügt der `__init__`-Methode ein Argument hinzu, so dass ihr Quadrate mit einer bestimmten Kantenlänge erstellen könnt, ohne die Kantenlänge nach der Erstellung eines Quadrats festlegen zu müssen:

```
>>> class Square:
...     def __init__(self, length):
...         self.length = length
...     def circumference(self):
...         return 4 * self.length
...
```

Warnung: `self.length` und `length` sind nicht dasselbe!

- `self.length` ist die Instanzvariable namens `length`
- `length` ist der lokale Funktionsparameter

In der Praxis würdet ihr den lokalen Funktionsparameter wahrscheinlich als `lng` oder `l` bezeichnen, um Verwechslungen zu vermeiden.

Mit dieser Definition von `Square` könnt ihr Quadrate mit beliebigen Kantenlängen mit einem Aufruf der Klasse `Square` erstellen. Im Folgenden wird ein Quadrat mit der Kantenlänge `3` erstellt:

```
...     s = Square(3)
```

Alle Standardfunktionen von Python – Standardargumente, zusätzliche Argumente, Schlüsselwortargumente usw. – können mit Methoden verwendet werden. Ihr hättet die erste Zeile von `__init__` wie folgt definieren können:

```
...     def __init__(self, length=1):
```

Dann würde der Aufruf von `Square` mit oder ohne zusätzliches Argument funktionieren; `Square()` würde ein Quadrat mit der Kantenlänge `1` und `Square(3)` ein Quadratmit der Kantenlänge `3` zurückgeben.

Bei einem Methodenaufruf `instance.method(arg1, arg2, ...)` wandelt Python diesen in einen normalen Funktionsaufruf um, indem es die folgenden Regeln anwendet:

1. Suche nach dem Methodennamen im Instanz-Namensraum. Wenn eine Methode für diese Instanz geändert oder hinzugefügt wurde, wird sie bevorzugt gegenüber Methoden in der Klasse aufgerufen.
2. Wenn die Methode nicht im Namensraum der Instanz gefunden wird, wird die Methode in der Klasse gesucht. In den vorangegangenen Beispielen ist `class` der `Square`-Typ der Instanz `s`.
3. Wenn die Methode immer noch nicht gefunden wurde, wird sie in einer Superklasse gesucht, s.a. *Vererbung*.
4. Wenn die Methode gefunden wurde, wird sie als normale Python-Funktion aufgerufen, wobei die Instanz als erstes Argument der Funktion verwendet und alle anderen Argumente im Methodenaufruf um ein Leerzeichen nach rechts verschoben werden. So wird `instance.method(arg1, arg2, ...)` zu `class.method(instance, arg1, arg2, ...)`.

13.3.1 Statische Methoden

Genau wie in Java könnt ihr statische Methoden aufrufen, auch wenn keine Instanz dieser Klasse erstellt wurde. Um eine statische Methode zu erstellen, verwendet den `@staticmethod`-*Dekorator*:

```
1  """circle module: contains the 'Circle' class"""
2
3
4  class Circle:
5      """Circle class.
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
```

Zeile 11
: definiert die Klassenvariable `circles` als zunächst leere Liste aller `Circle`-Instanzen.

Zeile 14
: fügt initialisierte `Circle`-Instanzen der `circles`-Liste hinzu.

```
>>> 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 Klassenmethoden

Klassenmethoden ähneln den statischen Methoden insofern, als sie aufgerufen werden können, bevor ein Objekt der Klasse instanziiert wurde. Allerdings wird den Klassenmethoden implizit die Klasse, zu der sie gehören, als erster Parameter übergeben:

```
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
```

Zeile 18
Der `@classmethod`-Dekorator wird vor der Methode `def` verwendet.

Zeile 19
Der Klassenparameter ist traditionell `cls`.

Zeile 22
Ihr könnt `cls` anstelle von `self.__class__` verwenden.

Durch die Verwendung einer Klassenmethode anstelle einer statischen Methode müsst ihr den Klassennamen nicht hart in `circumferences` codieren.

```
>>> import circle_cm
>>> c1 = circle_cm.Circle(1)
>>> c2 = circle_cm.Circle(2)
>>> circle_cm.Circle.circumferences()
9.424769999999999
```

13.4 Vererbung

Die Vererbung in Python ist einfacher und flexibler als die Vererbung in kompilierten Sprachen wie Java und C++, da die dynamische Natur von Python der Sprache nicht so viele Einschränkungen auferlegt.

Um zu sehen, wie Vererbung in Python verwendet wird, beginnen wir mit den Klassen `Square` und `Circle`, die wir früher bereits besprochen haben, und verallgemeinern sie.

Wenn wir diese Klassen nun in einem Zeichenprogramm verwenden wollen, müssen wir definieren, wo auf der Zeichenfläche sich eine Instanz befindet soll. Wir können dies tun, indem wir `x`- und `y`-Koordinaten für jede Instanz definieren:

```
>>> 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
...
```

Dieser Ansatz funktioniert, führt aber zu einer Menge sich wiederholenden Codes, wenn ihr die Anzahl der Form-Klassen erhöht, da ihr vermutlich wollt, dass jede Form diese Positionsangabe hat. Dies ist eine Standardsituation für die Verwendung von Vererbung in objektorientierten Sprachen. Anstatt die `x`- und `y`-Variablen in jeder Form-Klasse zu definieren, könnt ihr sie in eine allgemeine Form-Klasse abstrahieren und jede Klasse, die eine bestimmte Form definiert, von dieser allgemeinen Klasse erben lassen. In Python sieht diese Technik wie folgt aus:

```
>>> 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):
...         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
...
```

Zeilen 6 und 11
: `Square` und `Circle` erben von der `Form`-Klasse.

Zeilen 8 und 13
: rufen die `__init__`-Methode der `Form`-Klasse auf.

Es gibt im Allgemeinen zwei Anforderungen bei der Verwendung einer geerbten Klasse in Python, die ihr beide im Code der Klassen `Circle` und `Square` sehen könnt:

1. Die erste Anforderung besteht darin, die Vererbungshierarchie zu definieren, was ihr tut, indem ihr die Klassen, von denen geerbt wird, in Klammern unmittelbar nach dem Namen der Klasse angebt, die mit dem Schlüsselwort `class` definiert wird: `Circle` und `Square` erben beide von `Form`.
2. Das zweite Element ist der explizite Aufruf der `__init__`-Methode der geerbten Klasse. Dies erfolgt in Python nicht automatisch, sondern meist über die `super`-Funktion, genauer durch die Zeilen `super().__init__(x, y)`. Dieser Code ruft die Initialisierungsfunktion von `Form` mit der zu initialisierenden Instanz und den entsprechenden Argumenten auf. Andernfalls würden für die Instanzen von `Circle` und `Square` die Instanzvariablen `x` und `y` nicht gesetzt.

Die Vererbung kommt auch dann zum Tragen, wenn ihr versucht, eine Methode zu verwenden, die nicht in den Basisklassen, sondern in der Superklasse definiert ist. Um diesen Effekt zu sehen, definiert eine weitere Methode in der

Klasse `Form` mit dem Namen `move`, die eine Form in den `x`- und `y`-Koordinaten verschiebt. Die Definition für `Form` lautet nun:

```
>>> 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
...
```

Wenn ihr die Parameter `delta_x` und `delta_y` der Methode `move` in den `__init__`-Methoden von `Circle` und `Square` übernehmt, könnt ihr z.B. folgende interaktive Sitzung ausführen:

```
>>> c = Circle(3)
>>> c.move(4, 5)
>>> c.x
4
>>> c.y
5
```

Die Klasse `Circle` im Beispiel hat nicht direkt eine `move`-Methode in sich selbst definiert, aber da sie von einer Klasse erbt, die `move` implementiert, können alle Instanzen von `Circle` die `move`-Methode verwenden. In OOP-Begriffen könnte man sagen, dass alle Python-Methoden virtuell sind – D.H., wenn eine Methode in der aktuellen Klasse nicht existiert, wird die Liste der Oberklassen nach der Methode durchsucht und die erste gefundene verwendet.

13.5 Zusammenfassung

Die bisher angesprochenen Punkte, sind die Grundlagen der Verwendung von Klassen und Objekten in Python. Diese Grundlagen werde ich nun in einem einzigen Beispiel zusammenfassen:

1. Zunächst erstellen wir eine Basisklasse:

```
"""form module. Contains the classes Form, Square and Circle"""


class Form:
    """Form class: has method move"""

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Zeile 4
Die `__init__`-Methode benötigt eine Instanz (`self`) und zwei Parameter

Zeilen 5 und 6
Auf die beiden Instanzvariablen `x` und `y`, auf die über `self` zugegriffen wird.

Zeile 7
Die `move`-Methode benötigt eine Instanz (`self`) und zwei Parameter.

Zeilen 8 und 9
Instanzvariablen, die in der `move`-Methode gesetzt werden.

2. Als nächstes erstellt eine Unterklasse, die von der Basisklasse `Form` erbt:

```
11      def move(self, deltaX, deltaY):
12          self.x = self.x + deltaX
13          self.y = self.y + deltaY
14
15
16  class Square(Form):
17      """Square Class:inherits from Form"""
```

Zeile 11
Die Klasse `Square` erbt von der Klasse `Form`.

Zeile 13
`Square`'s `__init__` nimmt eine Instanz (`self`) und drei Parameter, alle mit Voreinstellungen.

Zeile 14
`__init__` von Circle verwendet `super()`, um `__init__` von `Form` aufzurufen.

3. Schließlich erstellen wir eine weitere Unterklasse, die zudem eine statische Methode enthält:

```
19      def __init__(self, length=1, x=0, y=0):
20          super().__init__(x, y)
21          self.length = length
22
23      def circumference(self):
24          return 4 * self.length
25
26
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):
34          super().__init__(x, y)
35          self.diameter = diameter
```

Zeilen 21 und 22
`pi` und `circles` sind Klassenvariablen für `Circle`.

Zeile 26
In der `__init__`-Methode fügt sich die Instanz in die Liste `circles` ein.

Zeilen 29 und 30
`circumferences` ist eine Klassenmethode und nimmt die Klasse selbst (`cls`) als Parameter.

Zeile 33
verwendet den Parameter `cls` für den Zugriff auf die Klassenvariable `circles`.

Jetzt könnt ihr einige Instanzen der Klasse `Circle` erstellen und sie analysieren. Da die `__init__`-Methode von `Circle` Standardparameter hat, könnt ihr einen Kreis erstellen, ohne irgendwelche Parameter anzugeben:

```
>>> import form
>>> c1 = form.Circle()
>>> c1.diameter, c1.x, c1.y
(1, 0, 0)
```

Wenn ihr Parameter angebt, werden diese verwendet, um die Werte der Instanz festzulegen:

```
>>> c2 = form.Circle(2, 3, 4)
>>> c2.diameter, c2.x, c2.y
(2, 3, 4)
```

Wenn ihr die `move()`-Methode aufruft, findet Python keine `move()`-Methode in der Klasse `Circle`, also wird in der Vererbungshierarchie nach oben gegangen und die `move()`-Methode von `Form` verwendet:

```
>>> c2.move(5, 6)
>>> c2.diameter, c2.x, c2.y
(2, 8, 10)
```

Ihr könnt auch die Klassenmethode `circumferences()` der Klasse `Circle` aufrufen, entweder über die Klasse selbst oder durch eine Instanz:

```
>>> form.Circle.circumferences()
9.424769999999999
>>> c2.circumferences()
9.424769999999999
```

13.6 Private Variablen und Methoden

Eine private Variable oder private Methode ist eine Variable, die außerhalb der Methoden der Klasse, in der sie definiert ist, nicht sichtbar ist. Private Variablen und Methoden sind aus zwei Gründen nützlich:

1. sie erhöhen die Sicherheit und Zuverlässigkeit, indem sie selektiv den Zugriff auf wichtige Teile der Implementierung eines Objekts verweigern
2. sie verhindern Namenskonflikte, die durch die Verwendung von Vererbung entstehen können.

Eine Klasse kann eine private Variable definieren und von einer Klasse erben, die eine private Variable mit demselben Namen definiert. Private Variablen erleichtern die Lesbarkeit von Code, da sie explizit angeben, was in einer Klasse nur intern verwendet wird. Alles andere ist die Schnittstelle der Klasse.

Die meisten Sprachen, die private Variablen definieren, tun dies durch die Verwendung des Schlüsselworts *private* o.Ä. (oder Ähnlichem). Die Konvention in Python ist einfacher und macht es auch leichter, sofort zu erkennen, was privat ist und was nicht. Jede Methode oder Instanzvariable, deren Name mit einem doppelten Unterstrich (__) beginnt, aber nicht endet, ist privat; alles andere ist nicht privat.

Betrachten wir als Beispiel die folgende Klassendefinition:

```
>>> 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):
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute '__y'
```

Die `print_y`-Methode ist nicht privat, und da sie sich in der `MyClass`-Klasse befindet, kann sie auf `__y` zugreifen und ausgeben:

```
>>> m.print_y()
2
```

Bemerkung: Der Mechanismus, der zur Gewährleistung der Privatsphäre verwendet wird, verfälscht den Namen privater Variablen und privater Methoden, wenn der Code zu Bytecode kompiliert wird. Konkret bedeutet dies, dass *`_ClassName`* dem Variablennamen vorangestellt wird:

```
>>> dir(m)
['_MyClass__y', '__class__', ...]
```

Damit soll also lediglich ein versehentlicher Zugriff verhindert werden.

13.7 `@property`-Dekorator

In Python könnt ihr direkt auf Instanzvariablen zugreifen, ohne zusätzliche Getter- und Setter-Methoden, die häufig in Java und anderen objektorientierten Sprachen verwendet werden. Dies macht das Schreiben von Python-Klassen sauberer und einfacher, aber in manchen Situationen kann die Verwendung von Getter- und Setter-Methoden auch nützlich sein. Nehmen wir an, dass ihr einen Wert benötigt, bevor ihr ihn in eine Instanzvariable setzt, oder ihr einfach den Wert eines Attributs herausfinden möchtet. In beiden Fällen würden Getter- und Setter-Methoden die Aufgabe erfüllen, allerdings um den Preis, dass der einfache Zugriff auf Instanzvariablen in Python verloren ginge.

Die Antwort ist die Verwendung einer *Property*. Diese kombiniert die Möglichkeit, den Zugriff auf eine Instanzvariable über Methoden wie Getter und Setter zu übergeben, mit dem einfachen Zugriff auf Instanzvariablen über die Punktnotation. Um eine *Property* zu erstellen, wird der `property`-Dekorator mit einer Methode verwendet, die den Namen der Eigenschaft hat:

```
    @property
    def length(self):
        return self.__length
```

Ohne Setter ist die *Property* `length` jedoch schreibgeschützt:

```
>>> s1 = form.Square()
>>> s1.length = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Um dies zu ändern, müsst ihr einen Setter hinzufügen:

```
    @length.setter
    def length(self, new_length):
        self.__length = new_length
```

Jetzt könnt ihr die Punkt-Notation verwenden, um die Eigenschaft `length` sowohl zu erhalten als auch zu setzen. Beachtet, dass der Name der Methode derselbe bleibt, aber der Dekorator ändert sich in den *Property*-Namen, in unserem Fall in `length.setter`:

```
>>> s1 = form.Square()
>>> s1.length = 2
>>> s1.circumference()
8
```

Ein großer Vorteil von Pythons Fähigkeit, Eigenschaften hinzuzufügen, besteht darin, dass ihr zu Beginn der Entwicklung mit einfachen alten Instanzvariablen arbeiten und dann nahtlos zu *Property*-Variablen wechseln könnt, wann immer und wo immer ihr dies benötigt, ohne den Client-Code zu ändern. Der Zugriff ist immer noch derselbe, unter Verwendung der Punktnotation.

13.8 Namensräume

Wenn ihr euch in der Methode einer Klasse befindet, habt ihr direkten Zugriff

1. auf den **lokalen Namensraum** mit den Parametern und Variablen, die in dieser Methode deklariert sind,
2. den **globalen Namensraum** mit Funktionen und Variablen, die auf Modulebene deklariert sind und
3. den **eingebauten Namensraum** mit den eingebauten Funktionen und eingebauten Exceptions.

Diese drei Namensräume werden in dieser Reihenfolge durchsucht.

Um die verschiedenen Namensräume in unserem Beispiel näher zu erläutern, haben wir unser bestehendes Modul so erweitert, dass deutlich wird, worauf innerhalb einer Methode zugegriffen werden kann: `form_ns.py`.

Einen Überblick über die Methoden, die in einem Namensraum verfügbar sind, erhaltet ihr mit

```
    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_
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪_', '__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']
```

Über die `self`-Variable habt ihr auch Zugriff auf

1. den **Namensraum der Instanz** mit
   - Instanzvariablen
   - privaten Instanzvariablen und
   - Instanzvariablen der Superklasse,
2. den **Namensraum der Klasse** mit
   - Methoden,
   - Klassenvariablen,
   - privaten Methoden und
   - privaten Klassenvariablen und
3. den **Namensraum der Superklasse** mit
   - Methoden der Superklasse und
   - Klassenvariablen der Superklasse.

Diese drei Namensräume werden ebenfalls in dieser Reihenfolge druchsucht.

Den Namensraum der Instanz könnt ihr nun z.B. analysieren mit der Methode `instance_variables`:

```
72      def instance_variables(self):
73          print(
74              "Instance variables self.__diameter, self.x, self.y:",
75              self.__diameter,
76              self.x,
77              self.y,
78          )
```

```
>>> import form_ns
>>> c1 = form_ns.Circle()
>>> c1.instance_variables()
Instance variables self.__diameter, self.x, self.y: 1 0 0
```

Bemerkung: Während ihr auf die Methode `move` der Superklasse `form` mit `self` zugreifen könnt, sind jedoch private Instanzvariablen, private Methoden und private Klassenvariablen der Superklasse so nicht zugänglich.

Wenn ihr nur Instanzen einer bestimmten Klasse ändern wollt, könnt ihr dies z.B. mit dem `Garbage Collector`:

```
>>> import forms
>>> c1 = forms.Circle()
>>> c2 = forms.Circle(2, 3, 4)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
>>> 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 Datentypen als Objekte

Inzwischen habt ihr die grundlegenden Python-*Datentypen* kennengelernt und wisst, wie ihr mit Hilfe von *Klassen* eure eigenen Datentypen erstellen könnt. Beachtet dabei, dass Python dynamisch typisiert ist, D.H., die Typen werden zur Laufzeit bestimmt, nicht zur Kompilierzeit. Dies ist einer der Gründe, warum Python so einfach zu benutzen ist. Ihr könnt einfach folgendes ausprobieren:

```
>>> type(3)
<class 'int'>
>>> type("Hello")
<class 'str'>
>>> type(["Hello", "Pythonistas"])
<class 'list'>
```

In diesen Beispielen seht ihr die eingebaute `type`-Funktion in Python. Sie kann auf jedes Python-Objekt angewendet werden und gibt den Typ des Objekts zurück. In diesem Beispiel sagt euch die Funktion, dass `3` ein `int` (Integer) ist, dass `'Hello'` ein `str` (String) und dass `['Hello', 'Pythonistas']` eine `list` (Liste) ist.

Von größerem Interesse dürfte jedoch die Tatsache sein, dass Python als Antwort auf die Aufrufe von `type` Objekte zurückgibt; `<class 'int'>`, `<class 'str'>` und `<class 'list'>` sind die Bildschirmdarstellungen der zurückgegebenen Objekte. Ihr könnt diese Python-Pbjekte also miteinander vergleichen:

```
>>> type("Hello") == type("Pythonistas!")
True
>>> type("Hello") == type("Pythonistas!") == type(["Hello", "Pythonistas"])
False
```

Mit dieser Technik könnt ihr U.A. eine Typüberprüfung in euren Funktions- und Methodendefinitionen durchführen. Die häufigste Frage zu den Typen von Objekten ist jedoch, ob ein bestimmtes Objekt eine Instanz einer Klasse ist. Ein Beispiel mit einer einfachen Vererbungshierarchie macht dies klarer:

1. Zunächst definieren wir zwei Klassen mit einer Vererbungshierarchie:

```
>>> class Form:
...     pass
...
>>> class Square(Form):
...     pass
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
...
>>> class Circle(Form):
...     pass
...
```

2. Nun könnt ihr eine Instanz `c1` der Klasse `Circle` erstellen:

```
>>> c1 = Circle()
```

3. Wie erwartet, gibt die type-Funktion auf `c1` aus, dass `c1` eine Instanz der Klasse `Circle` ist, die in Ihrem aktuellen `__main__` Namespace definiert ist:

```
>>> type(c1)
<class '__main__.Circle'>
```

4. Ihr könnt genau dieselben Informationen auch durch Zugriff auf das `__class__`-Attribut der Instanz erhalten:

```
>>> c1.__class__
<class '__main__.Circle'>
```

5. Ihr könnt auch explizit überprüfen, ob die beiden Klassenobjekte identisch sind:

```
>>> c1.__class__ == Circle
True
```

6. Zwei eingebaute Funktionen bieten jedoch benutzerfreundlichere Möglichkeit, die meisten der normalerweise benötigten Informationen zu erhalten:

   `isinstance()`
   : stellt fest, ob z.B. eine Klasse, die an eine Funktion oder Methode übergeben wird, vom erwarteten Typ ist.

   `issubclass()`

   : stellt fest, ob eine Klasse die Unterklasse einer anderen ist.

```
>>> 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

Die Verwendung von `type`, `isinstance()` und `issubclass()` macht es ziemlich einfach, die Vererbungshierarchie eines Objekts oder einer Klasse korrekt zu bestimmen. Python hat jedoch auch eine Funktion, die die Verwendung von Objekten noch einfacher macht: Duck-Typing:

> *„If it walks like a duck and it quacks like a duck, then it must be a duck."*

Dies bezieht sich auf Pythons Art und Weise zu bestimmen, ob ein Objekt der erforderliche Typ für eine Operation ist, wobei der Schwerpunkt auf der Schnittstelle eines Objekts liegt. Kurz gesagt müsst ihr euch in Python nicht um die Typüberprüfung von Funktions- oder Methodenargumenten und Ähnlichem kümmern, sondern euch stattdessen auf lesbaren und dokumentierten Code in Verbindung mit Tests verlassen, um sicherzustellen, dass ein Objekt bei Bedarf *„wie eine Ente quakt."*

Duck-Typing kann die Flexibilität von gut geschriebenem Code erhöhen und gibt euch in Kombination mit fortgeschrittenen objektorientierten Funktionen die Möglichkeit, Klassen und Objekte zu erstellen, die fast jede Situation abdecken. Solche speziellen Methoden sind Attribute einer Klasse mit besonderer Bedeutung für Python. Sie sind zwar als Methoden definiert, aber nicht dazu gedacht, sie direkt aufzurufen; stattdessen werden sie von Python automatisch als Reaktion auf eine Anforderung an ein Objekt dieser Klasse aufgerufen.

Eines der einfachsten Beispiele für eine spezielle Methode ist `object.__str__()`. Wenn es in einer Klasse definiert ist, wird das `__str__`-Methodenattribut jedes Mal aufgerufen, wenn eine Instanz dieser Klasse verwendet wird und Python eine benutzerlesbare Zeichenkettendarstellung dieser Instanz benötigt. Um dieses Attribut in Aktion zu sehen, verwenden wir erneut unsere `Form`-Klasse mit der Standardmethode `__init__` um Instanzen der Klasse zu initialisieren, sondern auch eine `__str__`-Methode um Zeichenketten zurückzugeben, die Instanzen in einem lesbaren Format darstellen:

```
>>> 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
```

Auch wenn unser spezielles `__str__`-Methodenattribut nicht von unserem Code explizit aufgerufen wurde, konnte es dennoch von Python verwendet werden, da Python weiß, dass das `__str__`-Attribut, falls vorhanden, eine Methode zur Umwandlung von Objekten in benutzerlesbare Zeichenketten definiert. Und genau dies zeichnet die speziellen Methodenattribute aus. So ist es z.B. oft eine gute Idee, das `__str__`-Attribut für eine Klasse zu definieren, damit ihr im Debugging-Code `print(instance)` aufrufen könnt und eine informative Aussage über euer Objekt zu erhalten.

Umgekehrt kann es jedoch auch verwundern, dass ein Objekttyp anders auf spezielle Methodenattribute reagiert. Daher verwende ich spezielle Methodenattribute meist nur in einer der folgenden beiden Fälle:

- in einer häufig verwendeten Klasse, meist für Sequenzen, die sich ähnlich wie ein in Python eingebauter Typ verhält, und die durch spezielle Methodenattribute nützlicher wird.
- in einer Klasse, die sich fast identisch zu einer eingebauten Klasse verhält, z.B. Listen, die als balancierte Bäume implementiert sind, um das Einfügen zu beschleunigen, kann ich die speziellen Methodenattribute definieren.

KAPITEL 14

Daten speichern und abrufen

Um Daten persistent zu speichern, kann ein Prozess verwendet werden, der sich *Serialisierung* oder *Marshalling* nennt. In ihm werden Datenstrukturen in eine lineare Form umgewandelt und gespeichert. Der umgekehrte Vorgang wird dann *Deserialisierung* oder *Unmarshalling* genannt. Python bietet in der Standardbibliothek mehrere Module, mit denen ihr Objekte serialisieren und deserialisieren könnt:

das marshal-Modul
: wird im Wesentlichen intern von Python genutzt und sollte nicht verwendet werden um Daten abwärtskompatibel zu speichern.

das *pickle*-Modul
: könnt ihr verwenden, wenn ihr weder ein lesbares Format noch Interoperabilität benötigt.

das json-Modul
: könnt ihr verwenden um Daten für verschiedene Sprachen in einer lesbaren Form auszutauschen.

das *xml*-Modul
: könnt ihr ebenfalls verwenden um Daten in verschiedene Sprachen in einer lesbaren Form auszutauschen.

14.1 Die Python-Datenbank-API

Die Python Database API (Application Programming Interface) definiert eine Standardschnittstelle für Python-Datenbankzugriffsmodule. Sie ist in **PEP 249** definiert und wird häufig verwendet, z.B. von *sqlite*, *psycopg*, and mysql-python.

14.2 SQLAlchemy

SQLAlchemy ist ein weit verbreitetes Datenbank-Toolkit. Es bietet nicht nur als ein ORM (Object Relational Mapper), sondern bietet auch eine allgemeine API zum Schreiben von datenbankagnostischem Code ohne SQL. Alembic basiert auf SQLAlchemy und dient als Datenbankmigrationswerkzeug.

14.3 NoSQL-Datenbanken

Es gibt Daten, die sich nur schwer in ein relationales Datenmodell übertragen lassen. Dann solltet ihr zumindest einen Blick auf NoSQL-Datenbanken werfen.

14.3.1 Dateisystem

Um mit Dateien zu arbeiten, müsst ihr häufig auch mit dem Dateisystem und den unterschiedlichen Konventionen je nach Betriebssystem interagieren. Hierfür zeige ich euch os und speziell os.path.

Pfade und Pfadnamen

Alle Betriebssysteme verweisen auf Dateien mit Zeichenketten, die als Pfadnamen bezeichnet werden. Python bietet eine Reihe von Funktionen, die euch helfen, einige Probleme zu lösen. Die Semantik von Pfadnamen ist auf allen Betriebssystemen sehr ähnlich, da das Dateisystem meist als Baumstruktur modelliert ist, wobei eine Festplatte die Wurzel und Ordner, Unterordner USW. die Zweige und Unterzweige darstellen; D.H., dass die meisten Betriebssysteme auf eine bestimmte Datei in sehr ähnlicher Art verweisen.

Verschiedene Betriebssysteme haben jedoch unterschiedliche Konventionen für Pfadnamen. Das Zeichen, das zur Trennung von aufeinanderfolgenden Datei- oder Verzeichnisnamen in einem Linux/macOS-Pfadnamen verwendet wird, ist `/`, während es in einem Windows-Pfadnamen \ ist. Außerdem hat das Linux-Dateisystem ein einziges Stammverzeichnis auf das durch ein `/`-Zeichen als erstes Zeichen im Pfadnamen verwiesen wird, während das Windows-Dateisystem für jedes Laufwerk ein eigenes Stammverzeichnis hat, das mit `C:\` USW. bezeichnet wird. Aufgrund dieser Unterschiede haben die Dateien auf den verschiedenen Betriebssystemen unterschiedliche Pfadnamen. Eine Datei namens `C:datamyfile` unter Windows könnte unter Linux und macOS `/data/myfile` sein. Python bietet Funktionen und Konstanten, mit denen ihr gängige Pfadnamenmanipulationen durchführen könnt, ohne sich um solche syntaktischen Details kümmern zu müssen. Mit ein wenig Sorgfalt können ihr eure Python-Programme so schreiben, dass sie unabhängig vom zugrunde liegenden Dateisystem korrekt ausgeführt werden.

Absolute und relative Pfade

Diese Betriebssysteme erlauben zwei Arten von Pfadnamen:

Absolute Pfadnamen
geben die genaue Position einer Datei im Dateisystem eindeutig an, indem sie den gesamten Pfad zu dieser Datei auflisten, beginnend mit dem Wurzelverzeichnis des Dateisystems.

Als Beispiele seien hier zwei absolute Windows-Pfadnamen genannt:

```
C:\Program Files\Python 3.9\
D:\backup\2022\06\
```

Und hier sind zwei absolute Linux-Pfadnamen und ein absoluter macOS-Pfadname:

```
/bin/python3
/cdrom/backup/2022/06/
/Applications/Python\ 3.10/
```

Relative Pfadnamen
geben die Position einer Datei relativ zu einem anderen Punkt im Dateisystem an, und dieser andere Punkt wird nicht im relativen Pfadnamen selbst angegeben.

Als Beispiel sei hier ein relativer Windows-Pfadnamen genannt:

```
save-data\filesystem.rst
```

... und hier ein relativer Linux/macOS-Pfadname:

```
save-data/filesystem.rst
```

Relative Pfade benötigen also einen Kontext, in dem sie verankert sind. Dieser Kontext wird in der Regel auf eine der beiden folgenden Arten bereitgestellt:

- Der relative Pfad wird an einen vorhandenen absoluten Pfad anzuhängt, wodurch ein neuer absoluter Pfad entsteht. Wenn ihr einen relativen Windows-Pfad *`Start Menu\Programs\Python 3.8`* und einen absoluten Pfad *`C:\Users\Veit`* habt, dann kann durch Anhängen des relativen Pfads ein neuer absoluter Pfad: `C:\Users\Veit\Start Menu\Programs\Python 3.8` erstellt werden. Wenn ihr denselben relativen Pfad an einen anderen absoluten Pfad anhängt (z.B. an `C:\Users\Tim`, so erhaltet ihr einen neuen Pfad, der sich auf ein anderes `HOME`-Verzeichnis (*`Tim`*) bezieht.
- Relative Pfade können auch einen Kontext erhalten durch den impliziten Verweis auf das aktuelle Arbeitsverzeichnis, also das Verzeichnis, in dem sich ein Python-Programm zum Zeitpunkt seiner Ausführung, befindet. Python-Befehle können implizit auf das aktuelle Arbeitsverzeichnis zurückgreifen, wenn ihnen ein relativer Pfad als Argument übergeben wird. Wenn ihr z.B. den Befehl `os.listdir('`*`RELATIVE/PATH`*`')` mit einem relativen Pfadargument verwendet, ist der Anker für diesen relativen Pfad das aktuelle Arbeitsverzeichnis, und das Ergebnis des Befehls ist eine Liste der Dateinamen in dem Verzeichnis, dessen Pfad durch Anhängen des aktuellen Arbeitsverzeichnisses an das relative Pfadargument gebildet wird.

  Das Verzeichnis, in dem sich eine Python-Datei befindet, wird als *current working directory* (ENGL. (englisch): aktuelles Arbeitsverzeichnis) bezeichnet. Dieses Verzeichnis wird sich meist von dem Verzeichnis unterscheiden, in dem sich der Python-Interpreter befindet. Um dies zu verdeutlichen, starten wir Python und verwenden den Befehl `os.getcwd()`, um das aktuelle Arbeitsverzeichnis von Python zu ermitteln:

  ```
  >>> import os
  >>> os.getcwd()
  '/home/veit'
  ```

Bemerkung: `os.getcwd()` wird als Funktionsaufruf ohne Argumente verwendet um zu verdeutlichen, dass der zurückgegebene Wert keine Konstante ist, sondern sich ändert, wenn ihr den Wert des aktuellen Arbeitsverzeichnisses ändert. Im obigen Beispiel ist das Ergebnis das Home-Verzeichnis auf einem meiner Linux-Rechner. Auf Windows-Rechnern würden zusätzliche Backslashes in den Pfad eingefügt: `C:\\Users\\Veit`, da Windows den Backslash \ als Pfadseparator verwendet, der in *Zeichenketten* jedoch eine andere Bedeutung hat.

Um euch die Inhalte des aktuellen Verzeichnisses anzeigen zu lassen, könnt ihr folgendes eingeben:

```
>>> os.listdir(os.curdir)
['.gnupg', '.bashrc', '.local', '.bash_history', '.ssh', '.bash_logout', '.
↪profile', '.idlerc', '.viminfo', '.config', 'Downloads', 'Documents', '.
↪python_history']
```

Ihr könnt jedoch auch in ein anderes Verzeichnis wechseln und euch dann das aktuelle Arbeitsverzeichnis ausgeben lassen:

```
>>> os.chdir("Downloads")
>>> os.getcwd()
'/home/veit/Downloads'
```

Pfadnamen ändern

Python bietet einige Möglichkeiten zum Ändern der Pfadnamen mit dem Submodul os.path, ohne explizit eine betriebssystemspezifische Syntax verwenden zu müssen.

`os.path.join()`
: konstruiert Pfadnamen für verschiedene Betriebssysteme, z.B. unter Windows:

```
>>> import os
>>> print(os.path.join("save-data", "filesystem.rst"))
save-data\filesystem.rst
```

Dabei werden die Argumente interpretiert als eine Reihe von Verzeichnis- oder Dateinamen, die zu einer einzigen Zeichenkette verbunden werden sollen, die vom zugrunde liegenden Betriebssystem als relativer Pfad verstanden wird. Unter Windows bedeutet dies, dass die Namen der Pfadkomponenten mit Backslashes (\) verbunden werden.

Wenn ihr das Gleiche unter Linux/macOS ausführt, erhaltet ihr hingegen als Separator /:

```
>>> import os
>>> print(os.path.join("save-data", "filesystem.rst"))
save-data/filesystem.rst
```

Ihr könnt mit dieser Methode also Dateipfade unabhängig vom Betriebssystem, auf dem euer Programm läuft, erstellen.

Die Argumente müssen auch nicht unbedingt einzelne Verzeichnis- oder Dateinamen sein; sie können auch Unterpfade sein, die dann zu einem längeren Pfadnamen zusammengefügt werden. Das folgende Beispiel veranschaulicht dies unter Windows, wobei entweder Schrägstriche (/) oder doppelte Backslashes (\\) in den Zeichenketten verwendet werden können:

```
>>> 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()`
: gibt ein Tupel mit zwei Elementen zurück, das den Basisnamen eines Pfades vom Rest des Pfades trennt, z.B. unter macOS:

```
>>> import os
>>> print(os.path.split(os.getcwd()))
('/home/veit/python-basics-tutorial-de', 'docs')
```

`os.path.basename()`
: gibt nur den Basisnamen des Pfades zurück:

```
>>> import os
>>> print(os.path.basename(os.getcwd()))
docs
```

`os.path.dirname()`
: gibt den Pfad bis zum Basisnamen zurück:

```
>>> import os
>>> print(os.path.dirname(os.getcwd()))
/home/veit/python-basics-tutorial-de
```

`os.path.splitext()`
: gibt die gepunktete Erweiterungsnotation aus, die von den meisten Dateisystemen verwendet wird, um den Dateityp anzugeben:

```
>>> import os
>>> print(os.path.splitext("filesystem.rst"))
('filesystem', '.rst')
```

Das letzte Element des zurückgegebenen Tupels enthält die gepunktete Erweiterung der angegebenen Datei.

`os.path.commonpath()`
: ist eine spezialisiertere Funktionen, um Pfadnamen zu manipulieren. Sie findet den gemeinsamen Pfad für eine Gruppe von Pfaden und ist so gut geeignet um das Verzeichnis der untersten Ebene zu finden, das jede Datei in einer Gruppe von Dateien enthält:

```
>>> import os
>>> print(os.path.commonpath(["save-data/filesystem.rst", "save-data/index.rst"]))
save-data
```

`os.path.expandvars()`
: erweitert Umgebungsvariablen in Pfaden:

```
>>> os.path.expandvars("$HOME/python-basics-tutorial-de")
'/home/veit/python-basics-tutorial-de'
```

Nützliche Konstanten und Funktionen

`os.name`
: gibt den Namen des Python-Moduls zurück, das importiert wurde, um die betriebssystemspezifischen Details zu handhaben, z.B.:

```
>>> import os
>>> os.name
'nt'
```

Bemerkung: Die meisten Versionen von Windows, mit Ausnahme von Windows CE, werden als `nt` identifiziert.

Auf macOS und Linux lautet die Antwort `posix`. Je nach Plattform könnt ihr mit dieser Antwort spezielle Operationen durchführen:

```
>>> import os
>>> if os.name == "posix":
...     root_dir = "/"
... elif os.name == "nt":
...     root_dir = "C:\\"
... else:
...     print("The operating system was not recognised!")
...
```

Informationen über Dateien erhalten

Dateipfade zeigen Dateien und Verzeichnisse auf eurer Festplatte an. Um mehr über sie zu erfahren, gibt es verschiedene Python-Funktionen, U.A.

`os.path.exists()`
: gibt `True` zurück, wenn sein Argument ein Pfad ist, der mit einem im Dateisystem existierenden Pfad übereinstimmt.

`os.path.isfile()`
: gibt `True` zurück, wenn und nur wenn der angegebene Pfad auf eine Datei hinweist, und gibt andernfalls `False` zurück, einschließlich der Möglichkeit, dass das Pfadargument auf nichts im Dateisystem hinweist.

`os.path.isdir()`
: gibt `True` zurück, wenn und nur wenn sein Pfadargument auf ein Verzeichnis hinweist; andernfalls gibt es `False` zurück.

Weitere ähnliche Funktionen stellen speziellere Abfragen bereit:

`os.path.islink()`
: gibt `True` zurück, wenn ein Pfad eine Datei angibt, die ein Link ist. Windows-Verknüfungsdateien mit der Endung `.lnk` sind jedoch in diesem Sinne keine echten Links und geben `False` zurück. Nur mit `mklink()` erstellte Links geben ebenfalls `True` zurück.

`os.path.ismount()`
: gibt unter `possix`-Dateisystemen `True` zurück, wenn der Pfad ein SOG. *Mount Point* oder Einhängepunkt ist.

`os.path.samefile()`
: gibt `True` zurück, wenn die beiden Pfadargumente auf dieselbe Datei zeigen.

`os.path.isabs()`
: gibt `True` zurück, wenn sein Argument ein absoluter Pfad ist; andernfalls wird `False` zurückgegeben.

`os.path.getsize()`
: gibt die Größe der Datei oder des Verzeichnisses an.

`os.path.getmtime()`
: gibt das Änderungsdatum der Datei oder des Verzeichnisses an.

`os.path.getatime()`
: gibt de letzte Zugriffszeit für eine Datei oder ein Verzeichnis an.

Weitere Dateisystemoperationen

Python verfügt über weitere, sehr nützlicher Befehle im `os`-Modul: Im Folgenden beschreibe ich nur einige betriebssystemübergreifende Operationen, es werden jedoch auch spezifischere Dateisystemfunktionen bereitgestellt.

`os.rename()`
: benennt oder verschiebt eine Datei oder ein Verzeichnis, z.B.

```
>>> os.rename("filesystem.rst", "save-data/filesystem.rst")
```

`os.remove()`
: löscht Dateien, z.B.

```
>>> os.remove("filesystem.rst")
```

`os.rmdir()`
: löscht ein leeres Verzeichnis. Um nicht leere Verzeichnisse zu entfernen, verwendet `shutil.rmtree()`; diese Funktion entfernt rekursiv alle Dateien in einem Verzeichnisbaum.

`os.makedirs()`
: erstellt ein Verzeichnis mit allen notwendigen Zwischenverzeichnissen, z.B.

```
>>> os.makedirs("save-data/filesystem")
```

Verarbeitung aller Dateien in einem Verzeichnis

Eine nützliche Funktion zum rekursiven Durchlaufen von Verzeichnisstrukturen ist die Funktion `os.walk()`. Mit ihr könnt ihr einen ganzen Verzeichnisbaum durchlaufen und für jedes Verzeichnis den Pfad dieses Verzeichnisses, eine Liste seiner Unterverzeichnisse und eine Liste seiner Dateien zurückgeben. Dabei kann sie drei optionale Argumente haben: `os.walk(directory, topdown=True, onerror=None, followlinks= False)`.

`directory`
: ist der Pfad des Startverzeichnisses

`topdown`
: auf `True` oder nicht vorhanden, verarbeitet die Dateien in jedem Verzeichnis vor den Unterverzeichnissen, was zu einer Auflistung führt, die oben beginnt und nach unten geht;

  auf `False` werden die Unterverzeichnisse jedes Verzeichnisses zuerst verarbeitet, was eine Durchquerung des Baums von unten nach oben ergibt.

`onerror`
: kann auf eine Funktion gesetzt werden, um Fehler zu behandeln, die aus Aufrufen von `os.listdir()` resultieren, die standardmäßig ignoriert werden. Üblicherweise wird symbolische Links nicht gefolgt, es sei denn, ihr gebt den Parameter `follow-links=True` an.

```
>>> 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
...
```

Zeile 4
prüft auf ein Verzeichnis namens `.ipynb_checkpoints`.

Zeile 5
entfernt `.ipynb_checkpoints` aus der Verzeichnisliste.

`shutil.copytree()` erstellt rekursiv Kopien aller Dateien eines Verzeichnisses und all seiner Unterverzeichnisse, wobei die Informationen über den Zugriffsmodus und den Status (D.H. die Zugriffs- und Änderungszeiten) erhalten bleiben. `shutil` verfügt auch über die bereits erwähnte Funktion `shutil.rmtree()` zum Entfernen eines Verzeichnisses und aller seiner Unterverzeichnisse sowie über mehrere Funktionen zum Erstellen von Kopien einzelner Dateien.

14.3.2 Das `pickle`-Modul

Python kann jede beliebige Datenstruktur in eine Datei schreiben, diese Datenstruktur wieder aus der Datei lesen und sie mit nur wenigen Befehlen neu erstellen. Diese Fähigkeit kann sehr nützlich sein, weil sie euch viele Seiten Code ersparen kann, die nichts anderes tut, als den Zustand eines Programms in eine Datei zu schreiben und diesen Zustand wieder einzulesen.

Python bietet diese Möglichkeit über das pickle-Modul. Pickle ist mächtig, aber einfach zu benutzen. Nehmen wir an, dass der gesamte Zustand eines Programms in drei Variablen gespeichert ist: `a`, `b` und `c`. Ihr könnt diesen Zustand wie folgt in einer Datei namens `data.pickle` speichern:

1. Importieren des `pickle`-Moduls

   ```
   >>> import pickle
   ```

2. Definieren verschiedener Daten

   ```
   >>> a = [1, 2.0, 3 + 4j]
   >>> b = ("character string", b"byte string")
   >>> c = {None, True, False}
   ```

3. Schreiben der Daten:

   ```
   >>> with open("data.pickle", "wb") as f:
   ...     pickle.dump(a, f)
   ...     pickle.dump(b, f)
   ...     pickle.dump(c, f)
   ...
   ```

   Es spielt keine Rolle, was in den Variablen gespeichert wurde. Der Inhalt kann so einfach sein wie Zahlen oder so

komplex wie eine Liste von Wörterbüchern, die Instanzen von benutzerdefinierten Klassen enthalten. `pickle.dump()` speichert alles.

Das Pickle-Modul kann fast alles auf diese Weise speichern. Es kann mit *Zahlen*, *Listen*, *Tupel*, *Dictionaries*, *Zeichenketten* und so ziemlich allem umgehen, was aus diesen Objekttypen besteht, also auch mit allen Klasseninstanzen. Es geht auch mit gemeinsam genutzten Objekten, zyklischen Referenzen und anderen komplexen Speicherstrukturen korrekt um, indem es gemeinsam genutzte Objekte nur einmal speichert und sie als gemeinsam genutzte Objekte wiederherstellt, nicht als identische Kopien.

4. Laden der gepickelten Daten:

   Diese Daten können bei einem späteren Programmlauf wieder eingelesen werden mit `pickle.load()`:

```
>>> with open("data.pickle", "rb") as f:
...     first = pickle.load(f)
...     second = pickle.load(f)
...     third = pickle.load(f)
...
```

5. Ausgeben der gepickelten Daten:

```
>>> print(first, second, third)
[1, 2.0, (3+4j)] ('character string', b'byte string') {False, None, True}
```

In den meisten Fällen werdet ihr jedoch nicht eure gesamten Daten in der gespeicherten Reihenfolge wiederherstellen wollen. Ein einfacher und effektiver Weg, nur die Daten von Interesse wiederherzustellen, besteht darin, eine Speicherfunktion zu schreiben, die alle zu speichernden Daten in einem Wörterbuch speichert und dann Pickle zum Speichern des Wörterbuchs verwendet. Anschließend könnt ihr eine ergänzende Wiederherstellungsfunktion verwenden, um das Wörterbuch wieder einzulesen und die Werte im Wörterbuch den entsprechenden Programmvariablen zuzuweisen. Wenn ihr diesen Ansatz mit dem vorherigen Beispiel verwendet, erhaltet ihr folgenden 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)
...
```

Anschließend könnt ihr gezielt die Daten aus `c` ausgeben mit

```
>>> with open("data.pickle", "rb") as f:
...     saved_data = pickle.load(f)
...     print(saved_data["c"])
...
{False, None, True}
```

Neben `pickle.dump()` und `pickle.load()` gibt es auch noch die Funktionen `pickle.dumps()` und `pickle.loads()`. Das angehängte `s` verweist darauf, dass diese Funktionen Strings verarbeiten.

Warnung: Obwohl die Verwendung eines gepickelten Objekts im vorherigen Szenario durchaus sinnvoll sein kann, solltet ihr euch auch der Nachteile von Pickles bewusst sein:

- Pickling ist weder besonders schnell noch platzsparend als Mittel zur Serialisierung. Selbst die Verwendung von json zur Speicherung serialisierter Objekte ist schneller und führt zu kleineren Dateien auf der Festplatte.

- Pickling ist nicht sicher, und das Laden eines Pickles mit bösartigem Inhalt kann zur Ausführung von beliebigem Code auf eurem Rechner führen. Daher solltet ihr das Pickling vermeiden, wenn die Möglichkeit besteht, dass die Pickle-Datei für jemanden zugänglich ist, der sie verändern könnte.
- Pickle-Versionen sind nicht immer rückwärtskompatibel.

Siehe auch:

- Python-Module-Dokumentation
- Using Pickle

14.3.3 Das `xml`-Modul

Das XML-Modul wird mit Python mitgeliefert. Im folgenden Abschnitt werden wir uns auf die zwei Untermodule minidom und ElementTree.

Arbeiten mit `minidom`

Im folgenden Beispiel analysieren wir `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>
      <date>2020-04-13</date>
   </book>
</catalog>
```

1. Hierzu impportieren wir zunächst das `minidom`-Modul und geben ihm denselben Namen, damit es leichter referenziert werden kann:

```
import xml.dom.minidom as minidom
```

2. Anschließend definieren wir die Methode `getTitles` und erfassen mit der Methode `getElementsByTagName` die gewünschten XML-Tags:

```
def getTitles(xml):
    """
    Print all titles found in books.xml
    """
    doc = minidom.parse(xml)
    node = doc.documentElement
    books = doc.getElementsByTagName("book")
```

3. Dann erstellen wir eine leere Liste namens `titles`, die mit den Titelobjekten gefüllt wird:

```
    titles = []
    for book in books:
        titleObj = book.getElementsByTagName("title")[0]
        titles.append(titleObj)
```

4. Nun wird in verschachtelten `for`-Schleifen der Titel ausgegeben:

```
    for title in titles:
        nodes = title.childNodes
        for node in nodes:
            if node.nodeType == node.TEXT_NODE:
                print(node.data)
```

5. Schließlich setzen wir die `__name__`-Variable noch wie `__main__` gesetzt, sodass das Modul wie das Hauptprogramm ausgeführt werden kann. Anschließend wenden wir unsere `getTitles`-Methode auf unsere `books.xml`-Datei an:

```
if __name__ == "__main__":
    document = "books.xml"
    getTitles(document)
```

Parsen mit ElementTree

1. Importieren von `cElementTree`:

```
import xml.etree.cElementTree as ET
```

Bemerkung: `cElementTree` ist in C geschrieben und ist erheblich schneller als `ElementTree`.

---

2. Anschließend definieren wir die Methode `parseXML` und das Wurzelelement `root`:

```
def parseXML(xml_file):
    """
    Parse XML with ElementTree
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    """
    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. Ausgeben der XML-Kindelemente von 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. Inhalte der Kindelemente mit `iter` ausgeben:

```
    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 Das `sqlite`-Modul

Die wichtigsten Eigenschaften von SQLite sind:

- in sich geschlossen
- serverlos
- konfigurationsfrei
- transaktional

SQLite wird verwendet, um Daten lokal zu speichern, z.B. in Mobiltelefonen (Android, iOS) und in Browsern (Firefox, Safari, Chrome) sowie in vielen anderen Anwendungen.

Siehe auch:

- sqlite home
- sqlite3 — DB-API 2.0 interface for SQLite databases
- W3Schools SQL tutorial

14.3.5 Erstellen einer Datenbank

1. Importiert das sqlite-Modul:

```
import sqlite3
```

2. Erstellt eine Datenbank:

```
conn = sqlite3.connect("library.db")

cursor = conn.cursor()
```

3. Erstellt eine Tabelle

```
cursor.execute(
    """CREATE TABLE books
                (title text, language text, author text, license text,
                        release_date text)
            """
)
```

14.3.6 Daten erstellen

1. Einfügen eines Datensatzes in die Datenbank:

```
cursor.execute(
    """INSERT INTO books
                VALUES ('Python basics', 'en', 'Veit Schiele', 'BSD',
                        '2021-10-28')"""
```

2. Daten in der Datenbank speichern:

```
conn.commit()
```

3. Fügt mehrere Datensätze mit der sichereren ?-Methode ein, wobei die Anzahl der ? der Anzahl der Spalten entsprechen sollte:

```
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 Daten aus csv erstellen

1. Die Module sqlite und csv importieren

```
import csv
import sqlite3
```

2. Zeigen auf die Bibliotheksdatenbank

```
conn = sqlite3.connect("library.db")
cursor = conn.cursor()
```

3. Lest die csv-Datei und fügt die Datensätze in die Datenbank ein:

```
with open("books.csv", encoding="utf-8") as f:
    reader = csv.reader(f, delimiter=",")
    cursor.executemany("INSERT INTO books VALUES (?,?,?,?,?)", reader)
```

4. Speichert die Daten in der Datenbank:

```
conn.commit()
```

14.3.8 Daten abfragen

1. Alle Datensätze eines Autors auswählen:

```
def select_all_records_from_author(cursor, author):
    print(f"All books from {author}:")
    sql = "SELECT * FROM books WHERE author=?"
    cursor.execute(sql, [author])
    for row in cursor.execute("SELECT * FROM books ORDER BY author"):
        print(row)
```

   Für die `print`-Ausgabe verwenden wir durch ein vorangestelltes `f` ein formatiertes Stringliteral oder f-string.

2. Alle Daten auswählen und nach Autor sortieren:

```
def select_all_records_sorted_by_author(cursor):
    print("Listing of all books sorted by author:")
    for row in cursor.execute("SELECT * FROM books ORDER BY author"):
        print(row)
```

3. Alle Titel auswählen, die Python enthalten:

```
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. Schließlich können die Daten abgefragt werden mit:

```
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 Daten aktualisieren

1. Methode zum ändern der Lizenz:

```
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. Aufruf der Methode:

```
update_license(old_name="BSD", new_name="BSD-3-Clause")
```

14.3.10 Daten löschen

1. Methode zum löschen aller Bücher einer bestimmten Sprache:

```
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. Aufrufen der Methode mit dem Parameter der zu löschenden Sprache:

```
delete_by_language(language="de")
```

14.3.11 Normalisieren der Daten

Unter Normalisierung wird die Aufteilung von Attributen oder Tabellenspalten in mehrere Relationen oder Tabellen verstanden, sodass keine Redundanzen mehr enthalten sind.

Beispiel

Im folgenden Beispiel normalisieren wir die Sprache, in der die Bücher veräffentlicht wurden.

1. Hierfür erstellen wir zunächst eine neue Tabelle `languages` mit den Spalten `id` und `language_code` anlegen:

```
cursor.execute(
    """CREATE TABLE languages
                (id INTEGER PRIMARY KEY AUTOINCREMENT,
                language_code VARCHAR(2))"""
```

2. Anschließend legen wir die Werte `de` und `en` in dieser Tabelle an:

```
cursor.execute(
    """INSERT INTO languages (language_code)
                VALUES ('de')"""
)

cursor.execute(
    """INSERT INTO languages (language_code)
```

3. Da SQLite `MODIFY COLUMN` nicht unterstützt, legen wir nun eine temporäre Tabelle `temp` an mit allen Spalten aus `books` und einer Spalte `language_code`, die die Spalte `id` aus der Tabelle `languages` als Fremdschlüssel verwendet:

```
cursor.execute(
    """CREATE TABLE "temp" (
                "id" INTEGER,
                "title" TEXT,
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
                    "language_code" INTEGER REFERENCES languages(id),
                    "language" TEXT,
                    "author" TEXT,
                    "license" TEXT,
                    "release_date" DATE,
                    PRIMARY KEY("id" AUTOINCREMENT)
                    )"""
```

4. Nun übernehmen wir die Werte aus der `books`-Tabelle in die `temp`-Tabelle:

```
cursor.execute(
    """INSERT INTO temp (title,language,author,license,release_date)
                    SELECT title,language,author,license,release_date FROM books"""
```

5. Die Angabe der Sprache in `books` als `id` der Datensätze aus der `languages`-Tabelle in `temp` übernehmen.

```
cursor.execute(
    """UPDATE temp
                    SET language_code = 1
                    WHERE language = 'de'"""
)
```

6. Nun können wir die Spalte `languages` in der Tabelle `temp` löschen:

```
cursor.execute("""ALTER TABLE temp DROP COLUMN language""")
```

Bemerkung: Erst ab Python-Versionen ab 3.8, die nach dem 27. April 2021 veröffentlicht wurden, kann `DROP COLUMN` verwendet werden.

Bei älteren Python-Versionen müsste eine weitere Tabelle angelegt werden, die nicht mehr die Spalte `languages` enthält und anschließend die Datensätze aus `templ` in diese Tabelle eingefügt werden.

---

7. Auch die `books`-Tabelle kann nun gelöscht werden:

```
cursor.execute("""DROP TABLE books""")
```

8. Und schließlich kann die `temp`-Tabelle umbenannt werden in `books`:

```
cursor.execute("""ALTER TABLE temp RENAME TO books""")
```

14.3.12 Abfragen normalisierter Daten

1. Abfragen aller Bücher sortiert nach `language_id` und `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. Um nun nicht nur die ID der Sprachen zu erhalten sondern die zugehörigen Sprachcodes wird mit `JOIN` über die `id`-Spalte in der `languages`-Tabelle eine Verbindung zu den dort hinterlegten Sprachcodes hergestellt:

```
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 Das psycopg-Modul

1. Installiert das psycopg-Modul:

```
$ 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
```

```
C:> python -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. Importiert das psycopg-Modul:

```
import psycopg2
```

3. Erstellt eine Datenbank:

```
conn = psycopg2.connect(dbname="my_db", user="username")
cursor = conn.cursor()
```

4. Abragen der Datenbank:

```
cursor.execute("SELECT * FROM my_table")
row = cursor.fetchone()
```

5. Zeiger und Verbindung schließen:

```
cursor.close()
conn.close()
```

KAPITEL 15

dataclasses

dataclasses wurden in Python 3.7 eingeführt und sind ein spezieller Shortcut, mit der wir Klassen erstellen können, die Daten speichern. Python bietet einen speziellen *Dekorator*, wenn wir eine solche Klasse erstellen wollen.

Bemerkung: Für Tabellendaten verwende ich im Allgemeinen pandas Series oder DataFrames und wenn ich Matrizen mit Zahlen speichern muss, verwende ich Numpy.

Nehmen wir an, wir wollen eine Klasse speichern, die ein Item repräsentiert mit `summary`, `owner`, `state` und `id`. Wir können eine solche Klasse definieren mit:

```
>>> from dataclasses import dataclass
>>> @dataclass
... class Item:
...     summary: str = None
...     owner: str = None
...     state: str = "todo"
...     id: int = None
...
```

Der `@dataclass`-Dekorator erstellt die `__init__`- und `__repr__`-Methoden. Wenn ich mir die Instanz der Klasse ausgeben lasse, erhalte ich den Klassennamen und die Attribute:

```
>>> i1
Item(summary='My first item', owner='veit', state='todo', id=1)
```

Im Allgemeinen werden Datenklassen als syntaktischer Zucker für die Erstellung von Klassen, die Daten speichern, verwendet. Ihr könnt euren Klassen zusätzliche Funktionalität verleihen, indem ihr Methoden definiert. Wir werden der Klasse eine Methode hinzufügen, die ein Item-Objekt aus einem *Dict* erstellt:

```
>>> @dataclass
... class Item:
...     ...
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
...     @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)
```

KAPITEL 16

Testen

Grundsätzlich wird zwischen statischen und dynamischen Testverfahren unterschieden.

Statische Testverfahren
werden verwendet um den Quellcode zu überprüfen, wobei dieser jedoch nicht ausgeführt wird. Sie unterteilen sich in

- Reviews und
- Statische Code-Analyse

  Es gibt diverse Python-Pakete, die euch bei der statischen Code-Analyse unterstützen können, u.a. flake8, Pysa und Wily.

Dynamische Testverfahren
dienen dem Auffinden von Fehlern beim Ausführen des Quellcodes. Dabei wird zwischen Whitebox- und Backbox-Tests unterschieden.

Whitebox-Tests
werden unter Kenntnis des Quellcodes und der Software-Struktur entwickelt. In Python stehen euch verschiedene Module zur Verfügung:

Unittest
unterstützt euch bei der Automatisierung von Tests.

Mock
erlaubt euch das Erstellen und Verwenden von Mock-Objekten.

Doctest
ermöglicht das Testen von in Python Docstrings geschriebenen Tests.

tox
ermöglicht das Testen in verschiedenen Umgebungen.

Blackbox-Tests
werden ohne Kenntnis des Quellcodes entwickelt. Neben *Unittest* kann in Python auch *Hypothesis* für solche Tests verwendet werden.

Siehe auch:

- Python Testing and Continuous Integration

16.1 Unittest

unittest unterstützt euch bei der Testautomatisierung mit gemeinsam genutztem Setup- und TearDown-Code sowie der Aggregation und Unabhängigkeit von Tests.

Hierfür liefert es die folgenden Testkonzepte:

Test Case (Testfall)
testet eine einzelnes Szenario.

Test Fixture (Prüfvorrichtung)
ist eine konsistente Testumgebung.

Siehe auch:

- pytest fixtures
- About fixtures
- Fixtures reference
- How to use fixtures

Test Suite
ist eine Sammlung mehrerer *Test Cases*.

Test Runner
durchläuft eine *Test Suite* und stellt die Ergebnisse dar.

16.1.1 Beispiel

Angenommen, ihr habt im Modul `test_arithmetic.py` die folgende Methode zum Hinzufügen implementiert:

```
def add(x, y):
    """
    >>> add(7,6)
    13
    """
    return x + y
```

... dann könnt ihr diese Methode mit einem Unittest testen.

1. Dazu müsst ihr zunächst euer Modul und das Unittest-Modul importieren:

```
import unittest
class TestArithmetic(unittest.TestCase):
```

2. Anschließend könnt ihr eine Testmethode schreiben, die eure Additionsmethode veranschaulicht:

```
class TestArithmetic(unittest.TestCase):
    def test_addition(self):
        self.assertEqual(arithmetic.add(7, 6), 13)

```

3. Damit die Unittests auch in andere Module importiert werden können, solltet ihr die folgenden Zeilen hinzufügen:

```
23 if __name__ == "__main__":
24     unittest.main()
```

4. Schließlich können alle Tests in `test_arithmetic.py` ausgeführt werden:

```
$ bin/python test_arithmetic.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

```
C:> python test_arithmetic.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

… oder etwas ausführlicher:

```
$ 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
```

Siehe auch:

- unittest — Unit testing framework

16.2 Beispiel: SQLite-Datenbank testen

1. Zum Testen, ob die Datenbank `library.db` mit `create_db.py` angelegt wurde, importieren wir neben sqlite3 und unittest auch noch `create_db.py` und os:

```
import os
import sqlite3
import unittest

import create_db
```

2. Anschließend definieren wir zunächst eine Testklasse `TestCreateDB`:

```
class TestCreateDB(unittest.TestCase):
```

3. In ihr definieren wir dann die Testmethode `test_db_exists`, in der wir mit `assert` die Annahme treffen, dass die Datei in os.path existiert:

```
    def test_db_exists(self):
        assert os.path.exists("library.db")
```

4. Nun überprüfen wir auch noch, ob die Tabelle `books` angelegt wurde. Hierfür versuchen wir, die Tabelle erneut anzulegen und erwarten mit `assertRaises`, dass `sqlite` mit einem `OperationalError` beendet wird:

```
    def test_table_exists(self):
        with self.assertRaises(sqlite3.OperationalError):
            create_db.cursor.execute("CREATE TABLE books(title text)")
```

5. Weitere Tests wollen wir nicht an einer Datenbank im Dateisystem durchführen sondern in einer SQLite-Datenbank im Arbeitsspeicher:

```
class TestCommands(unittest.TestCase):
    def setUp(self):
        self.conn = sqlite3.connect(":memory:")
        self.cursor = self.conn.cursor()
```

Siehe auch:

Weitere Beispiele zum Testen eurer SQLite-Datenbankfunktionen findet ihr in der SQLite Testsuite test_sqlite3.

16.3 Doctest

Das Python-Modul doctest prüft, ob die in einem Docstring angegebenen Tests erfüllt sind.

1. In `arithmetic.py` könnt ihr folgenden Docstring hinzufügen:

```
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):
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
      File "<stdin>", line 1, in <module>
    ZeroDivisionError: division by zero
    """
```

2. Anschließend könnt ihr ihn testen mit:

```
$ 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)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
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. Damit die Doctests auch in andere Module importiert werden können, solltet ihr die folgenden Zeilen hinzufügen:

```
38 if __name__ == "__main__":
39     import doctest
40
41     doctest.testmod(verbose=True)
```

16.4 Hypothesis

Hypothesis ist eine Bibliothek, mit der ihr Tests schreiben könnt, die aus einer Quelle von Beispielen parametrisiert werden. Anschließend werden einfache und verständliche Beispiele generiert, die dazu verwendet werden können, eure Tests fehlschlagen zu lassen und Fehler mit wenig Aufwand zu finden.

1. Installiert Hypothesis:

```
$ bin/python -m pip install hypothesis
```

```
C:> Scripts\python -m pip install hypothesis
```

   Alternativ kann Hypothesis auch mit Erweiterungen installiert werden, z.B.:

```
$ bin/python -m pip install hypothesis[numpy,pandas]
```

```
C:> Scripts\python -m pip install hypothesis[numpy,pandas]
```

2. Schreibt einen Test:

   1. Importe:

```
import pytest
from hypothesis import given
from hypothesis.strategies import floats, lists
```

   2. Testen:

```
@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. Test durchführen:

```
$ 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]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    @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 =============================
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
FAILED test_hypothesis.py::test_mean - assert inf <= 1.7976931348623157e+308
============================== 1 failed in 0.44s ===============================
```

Siehe auch:

Hypothesis for the Scientific Stack

16.5 pytest

pytest ist eine Alternative zu Pythons *Unittest*-Modul, die das Testen noch weiter vereinfacht.

16.5.1 Merkmale

- Ausführlichere Informationen über fehlgeschlagene `assert`-Anweisungen
- Automatische Erkennung von Testmodulen und -Funktionen
- Modulare Fixtures für die Verwaltung von kleinen oder parametrisierten, langlebigen Testressourcen
- Kann auch Unittests ohne Voreinstellungen ausführen
- Umfangreiche Plugin-Architektur, mit über 800 externen Plugins

16.5.2 Installation

Ihr könnt pytest in *virtuellen Umgebungen* installieren mit:

```
$ 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
```

Beispiele

Ihr könnt einfach eine Datei `test_one.py` anlegen mit folgendem Inhalt:

```
def test_sorted():
    assert sorted([4, 2, 1, 3]) == [1, 2, 3, 4]
```

Die Funktion `test_sorted()` wird von pytest als Testfunktion erkannt, weil sie mit `test_` beginnt und in einer Datei steht, die mit `test_` beginnt. Wenn der Test ausgeführt wird, bestimmt die `assert`-Anweisung, ob der Test erfolgreich war oder nicht. `assert` ist ein in Python eingebautes Schlüsselwort und löst eine `AssertionError`-Ausnahme aus, wenn der Ausdruck nach `assert` falsch ist. Jede nicht abgefangene Ausnahme, die innerhalb eines Tests ausgelöst wird, führt dazu, dass der Test fehlschlägt.

pytest ausführen

```
$ cd docs/test/pytest
$ pytest test_one.py
============================= test session starts ==============================
...
collected 1 item

test_one.py .                                                            [100%]

============================== 1 passed in 0.00s ===============================
```

Der Punkt hinter `test_one.py` bedeutet, dass ein Test durchgeführt und bestanden wurde. `[100%]` ist eine Prozentanzeige, die angibt, wie viele Tests der Testsitzung bisher durchgeführt wurden. Da es nur einen Test gibt, entspricht ein Test 100% der Tests. Wenn ihr mehr Informationen benötigt, könnt ihr `-v` oder `--verbose` verwenden:

```
$ 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` schlägt hingegen fehl:

```
$ 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 ===============================
```

Der fehlgeschlagene Test, `test_in`, erhält einen eigenen Abschnitt, um uns zu zeigen, warum er fehlgeschlagen ist. Und `pytest` sagt uns genau, was der erste Fehler ist. Dieser zusätzliche Abschnitt wird Traceback genannt. Das sind schon eine Menge Informationen, aber es gibt eine Zeile, die besagt, dass wir mit `-v` den kompletten Diff erhalten. Lasst uns das tun:

```
$ pytest -v test_two.py
============================= test session starts ==============================
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
...
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` fügt +- und --Zeichen hinzu, um uns genau die Unterschiede zu zeigen.

Bisher haben wir `pytest` mit dem Befehl `pytest` *`FILE`*`.py` ausgeführt. Lasst uns `pytest` nun auf ein paar weitere Arten laufen. Wenn ihr keine Dateien oder Verzeichnisse angebt, sucht `pytest` nach Tests im aktuellen Arbeitsverzeichnis und in Unterverzeichnissen; genauer wird nach `.py` Dateien gesucht, die mit `test_` beginnen oder mit `_test` enden. Wenn ihr pytest im Verzeichnis `docs/test/pytest` ohne Optionen startet, werden zwei Dateien mit Tests ausgeführt:

```
$ 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 =========================
```

Ich habe auch die Option `--tb=no` verwendet, um die Rückverfolgung (engl.: Traceback) abzuschalten, da wir die vollständige Ausgabe im Moment nicht wirklich brauchen.

Wir können auch eine Testfunktion innerhalb einer Testdatei angeben, die ausgeführt werden soll, indem wir `::test_`*`name`* zum Dateinamen hinzufügen:

```
$ pytest -v test_one.py::test_sorted
============================= test session starts ==============================
...
collected 1 item
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
test_one.py::test_sorted PASSED                                          [100%]

============================== 1 passed in 0.00s ===============================
```

Testergebnisse

Zu den möglichen Ergebnissen einer Testfunktion gehören:

PASSED (.)
: Der Test wurde erfolgreich durchgeführt.

FAILED (F)
: Der Test wurde nicht erfolgreich durchgeführt.

SKIPPED (s)
: Der Test wurde übersprungen.

XFAIL (x)
: Der Test sollte nicht bestehen, wurde aber durchgeführt und ist fehlgeschlagen.

XPASS (X)
: Der Test wurde mit `xfail` markiert, aber er lief und bestand.

ERROR (E)
: Eine Ausnahme ist während der Ausführung einer *Test-Fixtures* aufgetreten, nicht aber während der Ausführung einer Testfunktion.

Test-Funktionen schreiben

`assert`-Anweisungen

Wenn ihr Testfunktionen schreibt, ist die normale pytest-`assert`-Anweisung euer wichtigstes Werkzeug. Die Einfachheit dieser Anweisung bringt viele Entwickler dazu, pytest gegenüber anderen Frameworks zu bevorzugen. Im Folgenden findet ihr eine Liste einiger `assert`-Formen und `assert`-Hilfsfunktionen von *Unittest*:

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*`)`

Mit pytest könnt ihr `assert` *AUSDRUCK* mit einem beliebigen Ausdruck verwenden. Wenn der Ausdruck bei einer Konvertierung in einen booleschen Wert zu `False` ausgewertet würde, würde der Test fehlschlagen.

pytest enthält eine Funktion namens `assert rewriting`, die `assert`-Aufrufe abfängt und sie durch etwas ersetzt, das euch mehr darüber sagen kann, warum eure Annahmen fehlgeschlagen sind. Sehen wir uns an, wie hilfreich dieses Rewriting ist, indem wir uns einen fehlgeschlagenen `assert`-Test ansehen:

```
def test_equality_fails():
    i1 = Item("do something", "veit")
    i2 = Item("do something else", "veit")
    assert i1 == i2
```

Dieser Test schlägt fehl, aber interessant sind die Traceback-Informationen:

```
$ 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 ===============================
```

Das sind eine Menge Informationen:

Für jeden fehlgeschlagenen Test wird die genaue Zeile des Fehlers mit einem > angezeigt, das auf den Fehler verweist.

Die E-Zeilen zeigen Ihnen zusätzliche Informationen über den `assert`-Fehler, damit ihr herausfinden könnt, was falsch gelaufen ist. Ich habe absichtlich zwei Fehlanpassungen in `test_equality_fails()` eingegeben, aber nur die erste wurde angezeigt. Versuchen wir es noch einmal mit der `-vv`-Option, wie in der Fehlermeldung vorgeschlagen:

```
$ pytest -vv tests/test_item_fails.py
============================= test session starts ==============================
...
collected 1 item

tests/test_item_fails.py::test_equality_fails FAILED                     [100%]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
================================== 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='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 hat genau aufgelistet, welche Attribute übereinstimmen und welche nicht. Zudem wurden die genauen Abweichungen hervorgehoben.

Zum Vergleich können wir uns anzeigen lassen, was Python bei `assert`-Fehlern anzeigt. Um den Test direkt von Python aus aufrufen zu können, müssen wir einen Block am Ende von `tests/test_item_fails.py` einfügen:

```
if __name__ == "__main__":
    test_equality_fails()
```

Wenn wir den Test nun mit Python durchführen, erhalten wir folgendes Ergebnis:

```
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
```

Das sagt uns nicht viel. Die pytest-Ausgabe gibt uns viel mehr Informationen darüber, warum unsere Annahmen fehl-

geschlagen sind.

Fehlschlagen mit `pytest.fail()` und Exceptions

Das Fehlschlagen von Behauptungen ist die Hauptursache dafür, dass Tests fehlgeschlagen. Aber das ist nicht der einzige Weg. Ein Test schlägt auch fehl, wenn es eine nicht abgefangene *Exceptions* gibt. Das kann passieren, wenn

- eine `assert`-Anweisung fehlschlägt, was zu einer `AssertionError`-Exception führt,
- der Testcode `pytest.fail()` aufruft, was zu einer Exception führt, oder
- eine andere Exception ausgelöst wird.

Obwohl jede Exception einen Test fehlschlagen lassen kann, ziehe ich es vor, `assert` zu verwenden. In seltenen Fällen, in denen `assert` nicht geeignet ist, verwende ich meist `pytest.fail()`.

Hier ist ein Beispiel für die Verwendung der Funktion `fail()` von pytest, um einen Test explizit fehlschlagen zu lassen:

```
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!")
```

Die Ausgabe sieht wie folgt aus:

```
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 ===============================
```

Beim Aufruf von `pytest.fail()` oder dem Auslösen einer Exception, erhalten wir nicht das von pytest angebotene `assert`-Rewriting. Es gibt jedoch sinnvolle Gelegenheiten, `pytest.fail()` zu verwenden, wie z.B. in einem `assertion`-Hilfsprogramm.

Schreiben von assertion-Hilfsfunktionen

Eine assertion-Hilfsfunktion dient dazu, eine komplizierte assertion-Prüfung zu verpacken. Ein Beispiel: Die Datenklasse Item ist so eingerichtet, dass zwei Items mit unterschiedlichen IDs trotzdem Gleichheit berichten. Wenn wir eine strengere Prüfung wünschen, könnten wir eine Hilfsfunktion namens assert_ident wie folgt schreiben:

```
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)
```

Die assert_ident-Funktion setzt __tracebackhide__ = True. Die Folge ist, dass fehlgeschlagene Tests nicht in den Traceback aufgenommen werden. Das normale assert i1 == i2 wird dann verwendet, um alles außer id auf Gleichheit zu prüfen.

Schließlich werden die IDs überprüft pytest.fail() verwendet, um den Test mit einer hilfreichen Meldung fehlschlagen zu lassen. Schauen wir uns an, wie das nach der Ausführung aussieht:

```
$ 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 ============================
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
FAILED tests/test_helper.py::test_ident_fail - Failed: The IDs do not match: 42 != 43
========================= 1 failed, 1 passed in 0.03s ==========================
```

Testen auf erwartete Exceptions

Wir haben uns angesehen, wie jede Exception einen Test zum Scheitern bringen kann. Was aber, wenn ein Teil des Codes, den wir testen, eine Exception auslösen soll? Hierfür verwenden wir `pytest.raises()`, um auf erwartete Exceptions zu testen. Ein Beispiel hierfür wäre die Items-API, die eine `ItemsDB`-Klasse hat, die ein Pfadargument benötigt.

```
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 ===============================
```

Hier habe ich das kürzere Traceback-Format `--tb=short` verwendet, weil wir nicht den vollständigen Traceback sehen müssen, um herauszufinden, welche Exception ausgelöst wurde.

Die Exception `TypeError` erscheint sinnvoll, da der Fehler beim Versuch auftritt, den benutzerdefinierten ItemsDB-Typ zu initialisieren. Wir können einen Test schreiben, um sicherzustellen, dass diese Exception ausgelöst wird, etwa so:

```
import pytest

from items.api import ItemsDB


def test_db_exists():
    with pytest.raises(TypeError):
        ItemsDB()
```

Die Anweisung `with pytest.raises(TypeError):` besagt, dass der nächste Codeblock eine `TypeError`-Exception auslösen soll. Wenn keine Ausnahme ausgelöst wird oder eine andere Ausnahme ausgelöst wird, schlägt der Test fehl.

Wir haben gerade in `test_db_exists()` den Typ der Exception überprüft. Wir können auch überprüfen, ob die Meldung korrekt ist, oder jeden anderen Aspekt der Exception, wie z.B. zusätzliche Parameter:

```
def test_db_exists():
    match_regex = "missing 1 .* positional argument"
    with pytest.raises(TypeError, match=match_regex):
        ItemsDB()
```

oder

```
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)
```

Testsuite strukturieren

Stellt sicher, dass die Assertions am Ende von Testfunktionen aufbewahrt werden. Diese Empfehlung ist so verbreitet, dass sie mindestens zwei Namen hat:

Arrange-Act-Assert (AAA)
: wurde als Teil der *testgetriebenen Entwicklung (TDD)* populär.

Given-When-Then (GWT)
: wird im Kontext verhaltensgetriebener Entwicklung (BDD) verwendet.

Die Aufteilung in diese frei Phasen hat viele Vorteile. Dies trennt die Teile

Given/Arrange
: Der Ausgangszustand. Hier richtet ihr Daten oder die Umgebung ein, um die Aktion vorzubereiten.

When/Act
: Eine Aktion wird ausgeführt. Dies ist der Schwerpunkt des Tests – das Verhalten, von dem wir sicherstellen wollen, dass es richtig funktioniert.

Then/Assert
: Ein erwartetes Ergebnis oder ein Endzustand sollte eintreten. Am Ende des Tests stellen wir sicher, dass die Aktion zu dem erwarteten Verhalten geführt hat.

Ein häufig anzutreffendes Gegenmuster ist das Muster *Arrange–Assert–Act–Assert–Act–Assert…*, bei dem eine Vielzahl von Aktionen, gefolgt von Zustands- oder Verhaltensprüfungen, einen Arbeitsablauf validieren. Dies erscheint vernünftig, bis der Test fehlschlägt. Jede der Aktionen könnte den Fehler verursacht haben, so dass sich der Test nicht auf das Testen eines bestimmten Verhaltens konzentriert. Oder es könnte die Einrichtung in *Anordnen* gewesen sein, die den Fehler verursacht hat. Dieses verschachtelte `assert`-Muster führt zu Tests, die schwer zu debuggen und zu warten sind. Das Festhalten an *Given–When–Then* oder *Arrange–Act–Assert* hält den Test fokussiert und macht ihn wartungsfreundlicher.

Wenden wir diese Struktur als Beispiel auf einen unserer ersten Tests an:

```
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:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        # THEN the result will be a string
        pytest.fail("The items are not identical!")
```

Die Struktur hilft dabei, die Testfunktionen zu organisieren und sich auf das Testen **eines** Verhaltens zu konzentrieren. Die Struktur hilft euch auch dabei, an andere Testfälle zu denken. Die Konzentration auf einen Ausgangszustand hilft euch, an andere Zustände zu denken, die für das Testen der gleichen Aktion relevant sein könnten. Ebenso hilft die Konzentration auf ein ideales Ergebnis dabei, an andere mögliche Ergebnisse zu denken, wie z.B. Ausfallzustände oder Fehlerzustände, die ebenfalls mit anderen Testfällen getestet werden sollten.

Tests mit Klassen gruppieren

Bislang haben wir Testfunktionen innerhalb von Testmodulen in einem Dateisystemverzeichnis geschrieben. Diese Strukturierung des Testcodes funktioniert eigentlich ganz gut und ist für viele Projekte ausreichend. pytest erlaubt uns jedoch auch, Tests mit Klassen zu gruppieren. Nehmen wir einige der Testfunktionen, die sich auf die Gleichheit der Items beziehen, und gruppieren sie in einer Klasse:

```
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
```

Der Code sieht so ziemlich genauso aus wie vorher, mit der Ausnahme, dass jede Methode ein anfängliches `self`-Argument haben muss. Wir können nun alle diese Methoden zusammen ausführen, indem wir die Klasse angeben:

```
$ 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 ===============================
```

Wir können immer noch zu einer einzigen Methode kommen:

```
$ pytest -v tests/test_classes.py::TestEquality::test_equality
============================== test session starts ==============================
...
collected 1 item
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
tests/test_classes.py::TestEquality::test_equality PASSED                [100%]

============================== 1 passed in 0.00s ===============================
```

Wenn ihr mit *Objektorientierung* und *Klassenvererbung* vertraut seid, könnt ihr Hierarchien von Testklassen für vererbte Hilfsmethoden verwenden. Ich empfehle euch, Testklassen auch in produktivem Testcode nur sparsam und hauptsächlich zur Gruppierung zu verwenden. Wenn ihr euch mit der Vererbung von Testklassen zu viel Mühe gebt, wird das zukünftig verwirrend werden.

Teilmenge von Tests ausführen

Im vorangegangenen Abschnitt haben wir Testklassen verwendet, um eine Teilmenge von Tests ausführen zu können. Die Ausführung einer kleinen Gruppe von Tests ist beim Debuggen sehr praktisch, oder wenn ihr die Tests auf einen bestimmten Abschnitt der Codebasis beschränken wollt, an dem ihr gerade arbeitet. pytest erlaubt euch, eine Teilmenge von Tests auf verschiedene Arten auszuführen:

Teilmenge	Syntax
Alle Tests in einem Verzeichnis	`pytest path`
Alle Tests in einem Modul	`pytest path/test_module.py`
Alle Tests in einer Klasse	`pytest path/test_module.py::TestClass`
Einzelne Testfunktion	`pytest path/test_module.py::test_function`
Einzelne Testmethode	`pytest path/test_module.py::TestClass::test_method`
Tests, die einem Namensmuster entsprechen	`pytest -k pattern`
Tests nach Marker	siehe *Markers*

Ob `pytest` euren Testcode findet, hängt von der Namensgebung ab:

- Testdateien sollten `test_something.py` oder `something_test.py`.
- Testmethoden und Funktionen sollten `test_SOMETHING` genannt werden.
- Testklassen sollten den Namen `TestSomething` tragen.

Tipp: Verwendet eine Verzeichnisstruktur, die der Art und Weise entspricht, wie ihr euren Code ausführen möchtet, denn es ist einfach, ein komplettes Unterverzeichnis auszuführen. So könnt ihr Features und Funktionen unterteilen oder Subsysteme als Grundlage nehmen oder euch an der Code-Struktur orientieren.

Ihr könnt auch `-k pattern` verwenden, um Verzeichnisse, Klassen oder Testpräfixe zu filtern, also z.B. alle Tests der Klasse `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 =======================
```

oder alle Tests mit `equality` im Namen:

```
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%]

=========================== 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 ===================
```

Eines davon ist leider unser Fehlerbeispiel. Wir können es beseitigen, indem wir den Ausdruck erweitern:

```
$ 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 ========================
```

Die Schlüsselwörter `and`, `not`, `or` und `()` sind erlaubt, um komplexe Ausdrücke zu erstellen. Hier ist ein Testlauf aller Tests mit oder „ids“ im Namen, aber nicht in der Klasse „TestEquality“:

```
$ 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 ========================
```

Die Keyword-Option `-k` bietet zusammen mit `and`, `not` und `or` eine große Flexibilität bei der Auswahl der Tests, die ihr ausführen möchtet. Dies erweist sich bei der Fehlersuche oder der Entwicklung neuer Tests als sehr hilfreich.

Tipp: Es ist eine gute Idee, Anführungszeichen zu verwenden, wenn ihr einen Test zur Ausführung auswählt, da die Bindestriche, Klammern und Leerzeichen die Shells durcheinander bringen können.

Test-Fixtures

Nachdem ihr nun pytest zum Schreiben und Ausführen von Testfunktionen verwendet habt, wollen wir uns nun den *Fixtures* zuwenden, die für die Strukturierung von Testcode für fast jedes nicht-triviale Softwaresystem unerlässlich sind. Fixtures sind Funktionen, die von pytest vor (und manchmal nach) den eigentlichen Testfunktionen ausgeführt werden. Der Code in der Fixture kann tun, was immer ihr wollt. Ihr könnt Fixtures verwenden, um einen Datensatz zu erhalten, mit dem die Tests arbeiten sollen. Ihr könnt Fixtures verwenden, um ein System in einen bekannten Zustand zu versetzen, bevor ein Test ausgeführt wird. Fixtures werden auch verwendet, um Daten für mehrere Tests bereitzustellen.

In diesem Kapitel lernt ihr, wie ihr Fixtures erstellen und mit ihnen arbeiten könnt. Ihr werden lernen, wie Ihr Fixtures strukturieren, um sowohl Setup- als auch Teardown-Code zu speichern. Ihr werdet `scope` verwenden, um Fixtures einmal über viele Tests laufen zu lassen, und lernen, wie Tests mehrere Fixtures verwenden können. Ihr werdet auch lernen, wie ihr die Codeausführung durch Fixtures und Testcode verfolgen könnt.

Doch bevor ihr euch mit Fixtures vertraut machen und sie zum Testen von Items verwendet, sehen wir uns zunächst ein kleines Beispiel-Fixture an und erfahren, wie Fixtures und Testfunktionen miteinander verbunden sind.

Erste Schritte mit Fixtures

Hier ist ein einfaches Fixture, das eine Zahl zurückgibt:

```
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
```

Der `@pytest.fixture()`-*Dekorator* wird verwendet, um pytest mitzuteilen, dass eine Funktion eine Fixture ist. Wenn ihr den Fixture-Namen in die Parameter-Liste einer Testfunktion aufnehmt, weiß pytest, dass die Funktion vor der Ausführung des Tests ausgeführt werden soll. Fixtures können Arbeit verrichten und auch Daten an die Testfunktion zurückgeben. In diesem Fall dekoriert `@pytest.fixture()` die Funktion `some_data()`. Der Test `test_some_data()` hat den Namen der Fixture, `some_data()` als Parameter. pytest erkennt dies und sucht nach einer Fixture mit diesem Namen.

Testfixtures in pytest beziehen sich auf den Mechanismus, der die Trennung von *Vorbereitungen für-* und *Aufräumen nach*-Code von euren Testfunktionen ermöglicht. pytest behandelt Exceptions während Fixtures anders als während einer Testfunktion. Eine `Exception` oder ein `assert`-Fehler oder ein Aufruf von `pytest.fail()`, die während des eigentlichen Testcodes auftritt, führt zu einem `Fail`-Ergebnis. Während einer Fixture wird die Testfunktion jedoch als `Error` gemeldet. Diese Unterscheidung ist hilfreich bei der Fehlersuche, wenn ein Test nicht bestanden wurde. Wenn ein Test mit `Fail` endet, liegt der Fehler irgendwo in der Testfunktion, wenn ein Test mit `Error` endet, liegt der Fehler irgendwo in einer Fixture.

Fixtures für Setup und Teardown verwenden

Fixtures werden uns beim Testen der Items-Anwendung eine große Hilfe sein. Die Items-Anwendung besteht aus einer API, die den Großteil der Arbeit und der Logik übernimmt, einem schlanken CLI und eine Datenbank. Der Umgang mit der Datenbank ist ein Bereich, in dem Fixtures eine große Hilfe sein werden:

```
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
```

Um `count()` aufrufen zu können, benötigen wir ein Datenbankobjekt, das wir durch den Aufruf von `items.ItemsDB(db_path)()` erhalten. Die Funktion `items.ItemsDB()` gibt ein `ItemsDB`-Objekt zurück. Der Parameter `db_path` muss ein `pathlib.Path`-Objekt sein, das auf das Datenbankverzeichnis zeigt. Zum Testen funktioniert ein temporäres Verzeichnis, das wir mit `tempfile.TemporaryDirectory()` erhalten.

Diese Testfunktion enthält jedoch einige Probleme: Der Code, um die Datenbank einzurichten, bevor wir `count()` aufrufen, ist nicht wirklich das, was wir testen wollen. Auch kann die `assert`-Anweisung nicht vor dem Aufruf von `db.close()` erfolgen, denn wenn die `assert`-Anweisung fehlschlägt, wird de Datenbankverbindung nicht mehr geschlossen. Diese Probleme lassen sich mit pytest-Fixture lösen:

```
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
```

Die Testfunktion selbst ist nun viel einfacher zu lesen, da wir die gesamte Datenbankinitialisierung in eine Fixture namens `items_db` ausgelagert haben. Die Fixture `items_db` bereitet den Test vor, indem sie die Datenbank bereitstellt und anschließend das Datenbankobjekt ausgibt. Erst dann wird der Test ausgeführt. Und erst nachdem der Test gelaufen ist, wird die Datenbank wider geschlossen.

Fixture-Funktionen werden vor den Tests ausgeführt, die sie verwenden. Wenn es in der Funktion einen `yield` gibt, wird dort angehalten, die Kontrolle an die Tests übergeben und in der nächsten Zeile fortgesetzt, nachdem die Tests abgeschlossen sind. Der Code oberhalb von `yield` ist *Setup* und der Code nach dem `yield` ist *Teardown*. Der *Teardown* wird garantiert ausgeführt, unabhängig davon, was während der Tests passiert.

In unserem Beispiel erfolgt `yield` innerhalb eines Kontextmanagers mit einem temporären Verzeichnis. Dieses Ver-

zeichnis bleibt bestehen, während das Fixture verwendet wird und die Tests laufen. Nach Beendigung des Tests wird die Kontrolle wieder an das Fixture übergeben, `db.close()` kann ausgeführt werden und der `with`-Block kann den Zugriff auf das Verzeichnis schließen.

Wir können Fixtures auch in mehreren Tests verwenden, z.B. 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()` verwendet dasselbe `items_db`-Fixture. Diesmal nehmen wir die leere Datenbank und fügen zwei Items hinzu, bevor wir die Anzahl überprüfen. Wir können `items_db` nun für jeden Test verwenden, der eine konfigurierte Datenbank benötigt. Die einzelnen Tests, wie `test_empty()` und `test_count()`, können kleiner gehalten werden und konzentrieren sich auf das, was wir wirklich testen wollen, und nicht auf *Setup* und *Teardown*.

Fixture-Ausführung mit `--setup-show` anzeigen

Da wir nun zwei Tests haben, die dieselbe Fixture verwenden, wäre es interessant zu wissen, in welcher Reihenfolge sie aufgerufen werden. pytest bietet die Kommandozeilen-Option `--setup-show`, das uns die Reihenfolge der Operationen von Tests und Fixtures anzeigt, einschließlich der Setup- und Teardown-Phasen der Fixtures:

```
$ pytest --setup-show tests/test_count.py
============================= test session starts ==============================
...
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 ===============================
```

Wir können sehen, dass unser Test läuft, umgeben von den `SETUP`- und `TEARDOWN`-Teilen der `items_db`-Fixture. Das `F` vor dem Namen der Fixture zeigt an, dass die Fixture den Funktionsumfang verwendet, D.H. die Fixture wird vor jeder Testfunktion aufgerufen, die sie verwendet, und danach wieder abgebaut. Schauen wir uns als nächstes den Funktionsumfang an.

Umfang einer Fixture festlegen

Jedes Fixture hat einen bestimmten Umfang, der die Reihenfolge der Ausführung von *Setup* und *Teardown* im Verhältnis zur Ausführung aller Testfunktionen, die das Fixture verwenden, festlegt. Der Geltungsbereich bestimmt, wie oft *Setup* und *Teardown* ausgeführt werden, wenn sie von mehreren Testfunktionen verwendet werden.

Wenn das Einrichten und Verbinden mit der Datenbank oder das Erzeugen großer Datensätze jedoch zeitaufwändig ist, kann es jedoch vorkommen, dass ihr dies nicht für jeden einzelnen Test ausführen wollt. Wir können einen Bereich so ändern, dass der langsame Teil nur einmal für mehrere Tests passiert. Ändern wir den Bereich unserer Fixture so, dass die Datenbank nur einmal geöffnet wird, indem `scope="module"` zum Fixture Decorator hinzugefügt wird:

```
@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 ===============================
```

Wir haben diese Einrichtungszeit für die zweite Testfunktion eingespart. Durch die Änderung des Modulumfangs kann jeder Test in diesem Modul, der die `items_db`-Fixture verwendet, dieselbe Instanz davon nutzen, ohne dass zusätzliche Einrichtungs- und Abbauzeit anfällt.

Der Fixture-Parameter `scope` erlaubt jedoch mehr als nur `module`:

scope-Werte	Beschreibung
`scope='fun`	Standardwert. Wird einmal pro Testfunktion ausgeführt.
`scope='cla`	Wird einmal pro Testklasse ausgeführt, unabhängig davon, wie viele Testmethoden die Klasse enthält.
`scope='mod`	Wird einmal pro Modul ausgeführt, unabhängig davon, wie viele Testfunktionen oder -methoden oder andere Fixtures im Modul es verwenden.
`scope='pac`	Wird einmal pro Paket oder Testverzeichnis ausgeführt, unabhängig davon, wie viele Testfunktionen oder -methoden oder andere Fixtures in dem Paket verwendet werden.
`scope='ses`	Wird einmal pro Sitzung ausgeführt. Alle Testmethoden und -funktionen, die ein Fixture mit Session-Scope verwenden, teilen sich einen Aufruf zum Einrichten und Abbauen.

Der Geltungsbereich wird also bei der Definition einer Fixture festgelegt und nicht an der Stelle, an der sie aufgerufen wird. Die Testfunktionen, die ein Fixture verwenden, steuern nicht, wie oft ein Fixture auf- und abgebaut wird.

Bei einer Fixture, die innerhalb eines Testmoduls definiert ist, verhalten sich die Session- und Package-Scopes genau wie die Module-Scopes. Um diese anderen Bereiche nutzen zu können, müssen wir eine `conftest.py`-Datei verwenden.

Gemeinsame Nutzung von Fixtures mit `conftest.py`

Ihr könnt Fixtures in einzelne Testdateien einfügen, aber um Fixtures für mehrere Testdateien freizugeben, müsst ihr eine `conftest.py`-Datei entweder im selben Verzeichnis wie die Testdatei, die sie verwendet, oder in einem übergeordneten Verzeichnis verwenden. Dabei ist die Datei `conftest.py` optional. Sie wird von pytest als ein *lokales Plugin* betrachtet und kann Hook-Funktionen und Fixtures enthalten. Beginnen wir damit, das `items_db`-Fixture aus `test_count.py` in eine `conftest.py`-Datei im selben Verzeichnis zu verschieben:

```
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()
```

Bemerkung: Fixtures können nur von anderen Fixtures desselben oder eines größeren Bereichs abhängen. Eine Fixture mit Funktionsumfang kann also von anderen Fixtures mit Funktionsumfang abhängen. Ein Function-Scope-Fixture kann auch von `class`-, `module`- und `session`-Scope-Fixtures abhängen, aber nicht umgekehrt.

Warnung: Obwohl `conftest.py` ein Python-Modul ist, sollte es nicht von Testdateien importiert werden. Die Datei `conftest.py` wird automatisch von pytest gelesen, so dass ihr nirgendwo `conftest` importieren müsst.

Finden, wo Fixtures definiert sind

Wir haben eine Fixture aus dem Testmodul in eine `conftest.py`-Datei verschoben. Wir können `conftest.py`-Dateien auf wirklich jeder Ebene unseres Testverzeichnisses haben. Die Tests können jede Fixture verwenden, die sich im selben Testmodul wie eine Testfunktion befindet, oder in einer `conftest.py`-Datei im selben Verzeichnis, oder auf jeder Ebene des übergeordneten Verzeichnisses bis hin zur Wurzel der Tests.

Das bringt ein Problem mit sich, wenn man sich nicht mehr daran erinnern kann, wo sich eine bestimmte Fixture befindet und man den Quellcode sehen möchte. Mit `pytest --fixtures` können wir uns anzeigen lassen, wo die Fixtures definiert sind:

```
pytest --fixtures
============================= test session starts ==============================
...
collected 10 items
cache -- .../_pytest/cacheprovider.py:532
    Return a cache object that can persist state between testing sessions.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
...
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 zeigt uns eine Liste aller verfügbaren Fixtures, die unser Test verwenden kann. Diese Liste enthält eine Reihe von eingebauten Fixtures, die wir uns in *Built-in Fixtures* ansehen werden, sowie Fixtures, die von *Plugins* bereitgestellt werden. Die Fixtures, die in `conftest.py`-Dateien gefunden werden, stehen am Ende der Liste. Wenn ihr ein Verzeichnis angebt, listet pytest die Fixtures auf, die für Tests in diesem Verzeichnis verfügbar sind. Wenn ihr den Namen einer Testdatei angebt, schließt pytest auch die in den Testmodulen definierten Fixtures ein.

Die Ausgabe von pytest enthält

- die erste Zeile des Docstrings der Fixture

  Durch Hinzufügen von `-v` wird der gesamte Docstring eingeschlossen.

- die Datei- und Zeilennummer, in der die Fixture definiert ist
- der Pfad, wenn die Fixture sich nicht im aktuellen Verzeichnis befindet

---

Bemerkung: Beachtet, dass wir für pytest 6.x `-v` verwenden müssen, um den Pfad und die Zeilennummern zu erhalten. Erst ab pytest 7 werden diese ohne weitere Option hinzugefügt.

Ihr könnt auch `--fixtures-per-test` verwenden, um zu sehen, welche Fixtures von jedem Test verwendet werden und wo die Fixtures definiert sind:

```
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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
============================ no tests ran in 0.00s =============================
```

In diesem Beispiel haben wir einen einzelnen Test angegeben: `test_count.py::test_empty`. Es können jedoch auch Dateien oder Verzeichnisse angegeben werden.

Mehrere Fixture-Levels verwenden

Unser Testcode ist momentan noch problematisch, da beide Tests davon abhängen, dass die Datenbank zu Beginn leer ist. Dieses Problem wird sehr deutlich, wenn wir einen dritten Test hinzufügen:

```
$ pytest test_count.py::test_count2
============================= test session starts ==============================
...
collected 1 item

test_count.py .                                                          [100%]

============================== 1 passed in 0.00s ===============================
```

Es funktioniert einzeln ausgeführt, aber nicht, wenn er nach `test_count.py::test_count` ausgeführt wird:

```
$ 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

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 =========================
```

Es gibt drei Items in der Datenbank, weil der vorherige Test bereits zwei Elemente hinzugefügte, bevor `test_count2` ausgeführt wurde. Tests sollten sich jedoch nicht auf die Ausführungsreihenfolge verlassen. `test_count2` ist nur erfolgreich, wenn er alleine ausgeführt wird, schlägt aber fehl, wenn er nach `test_count` ausgeführt wird.

Wenn wir immer noch versuchen wollen, mit einer offenen Datenbank zu arbeiten, aber alle Tests mit null Items in der Datenbank starten sollen, können wir das tun, indem wir eine weitere Fixture in `conftest.py` hinzufügen:

```
@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
```

Ich habe die alte `items_db` in `db` umbenannt und sie in den Session-Bereich verschoben.

Die `items_db`-Fixture hat `db` in ihrer Parameter-Liste, was bedeutet, dass sie von der `db`-Fixture abhängt. Außerdem ist `items_db` `function`-orientiert, was einen engeren Bereich als `db` darstellt. Wenn Fixtures von anderen Fixtures abhängen, können sie nur Fixtures verwenden, die den gleichen oder einen größeren Geltungsbereich haben.

Schauen wir mal, ob es funktioniert:

```
$ 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

============================== 3 passed in 0.00s ===============================
```

Wir sehen, dass die Einrichtung für `db` zuerst erfolgt und den Geltungsbereich der Session hat (vom `S`). Das Setup für `items_db` erfolgt als nächstes und vor jedem Test-Funktionsaufruf und hat den Geltungsbereich der Funktion (vom `F`). Außerdem werden alle drei Tests bestanden.

Die Verwendung von Fixtures für mehrere Stufen kann unglaubliche Geschwindigkeitsvorteile bieten und die Unabhängigkeit der Testreihenfolge wahren.

Mehrere Fixtures pro Test oder Fixture verwenden

Eine weitere Möglichkeit, mehrere Fixtures zu verwenden, besteht darin, mehr als eine in einer Funktion oder einem Fixture zu verwenden. Zum Beispiel können wir einige vorgeplante Items zusammenstellen, um sie in einem Fixture zu testen:

```
@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
```

Und auch Fixtures können mehrere andere Fixtures verwenden:

```
@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
```

Die Fixture `populated_db` muss im `function`-Bereich liegen, da sie `items_db` verwendet, das bereits im `function`-Bereich liegt. Wenn ihr versuchen solltet, `populated_db` in den `module`-Bereich oder einen größeren Bereich zu setzen, wird pytest einen Fehler ausgeben. Vergesst nicht, dass ihr, wenn ihr keinen Bereich angebt, Fixtures im `function`-Bereich erhaltet. Tests, die eine gefüllte Datenbank benötigen, können dies nun einfach tun mit

```
def populated(populated_db):
    assert populated_db.count() > 0
```

Wir haben gesehen, wie verschiedene Fixture-Scopes funktionieren und wie verschiedene Scopes in verschiedenen Fixtures genutzt werden können. Es kann jedoch vorkommen, dass ihr einen Bereich zur Laufzeit festlegen müsst. Das ist mit dynamischem Scoping möglich.

Fixture-Scope dynamisch festlegen

Nehmen wir an, wir haben die Fixtures so eingerichtet wie jetzt, mit `db` im `session`-Scope und `items_db` im `function`-Scope. Nun besteht jedoch die Gefahr, dass das `items_db`-Fixture leer ist, weil es `delete_all()` aufruft. Deshalb wollen wir eine Möglichkeit schaffen, die Datenbank für jede Testfunktion vollständig einzurichten, indem wir den Scope der `db`-Fixture zur Laufzeit dynamisch festlegen. Hierfür ändern wir zuerst den Scope von `db` in der `conftest.py`-Datei:

```
@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()
```

Anstelle eines bestimmten Bereichs haben wir einen Funktionsnamen eingegeben: `db_scope`. Nun müssen wir also noch diese Funktion schreiben:

```
def db_scope(fixture_name, config):
    if config.getoption("--fdb", None):
        return "function"
    return "session"
```

Es gibt viele Möglichkeiten, wie wir herausfinden können, welchen Bereich wir verwenden sollen. In diesem Fall habe ich mich für eine neue Kommandozeilenoption `--fdb` entschieden. Damit wir diese neue Option mit pytestverwenden können, müssen wir eine Hook-Funktion in der `conftest.py`-Datei schreiben, die ich in *Plugins* näher erläutern werde:

```
def pytest_addoption(parser):
    parser.addoption(
        "--fdb",
        action="store_true",
        default=False,
        help="Create new db for each test",
    )
```

Nach all dem ist das Standardverhalten dasselbe wie vorher, mit `db` im `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).
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        TEARDOWN F items_db
TEARDOWN S db

============================== 3 passed in 0.00s ===============================
```

Wenn wir jedoch die neue Option verwenden, erhalten wir eine db-Fixture im `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 ===============================
```

Die Datenbank wird nun vor jeder Testfunktion aufgebaut und danach wieder abgebaut.

`autouse` für Fixtures, die immer verwendet werden

Bisher wurden alle von Tests verwendeten Fixtures durch die Tests oder eine andere Fixture in einer Parameterliste benannt. Ihr könnt jedoch `autouse=True` verwenden, um ein Fixture immer laufen zu lassen. Dies eignet sich gut für Code, der zu bestimmten Zeiten ausgeführt werden soll, aber Tests sind nicht wirklich von einem Systemzustand oder Daten aus der Fixture abhängig, z.B.:

```
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

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 ===============================
```

Tipp: Das `autouse`-Feature sollte eher die Ausnahme als die Regel sein. Entscheidet euch für benannte Fixtures, es sei denn, ihr habt einen wirklich triftigen Grund, dies nicht zu tun.

Fixtures umbenennen

Der Name einer Fixture, der in der Parameterliste von Tests und anderen Fixtures aufgeführt ist, die diese Fixture verwenden, ist normalerweise derselbe wie der Funktionsname der Fixture. Pytest erlaubt jedoch das Umbenennen von Fixtures mit einem Namensparameter an `@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
```

Ein Fall, in dem eine Umbenennung sinnvoll sein kann, ist, wenn der naheliegendste Fixture-Name bereits als Variablen- oder Funktionsname existiert.

Built-in Fixtures

Die Wiederverwendung gemeinsamer Fixtures ist eine so gute Idee, dass pytest einige häufig verwendete Fixtures integriert hat. Die eingebauten Fixtures, helfen euch, einige sehr nützliche Dinge in euren Tests einfach und konsistent zu tun. Unter anderem enthält pytest eingebaute Fixtures, die mit temporären Verzeichnissen und Dateien umgehen, auf Kommandozeilenoptionen zugreifen, zwischen Testsitzungen kommunizieren, Ausgabeströme validieren, Umgebungsvariablen verändern und Warnungen abfragen können.

`tmp_path` und `tmp_path_factory`

Die Fixtures tmp_path und tmp_path_factory werden verwendet, um temporäre Verzeichnisse zu erstellen. Die Fixture `tmp_path` für den `function`-Scope gibt eine `pathlib.Path`-Instanz zurück, die auf ein temporäres Verzeichnis verweist, das während des Tests und etwas länger bestehen bleibt. Die `tmp_path_factory` für eine `session`-Scope-Fixture gibt ein `TempPathFactory`-Objekt zurück. Dieses Objekt hat eine `mktemp()`-Funktion, die `Path`-Objekte zurückgibt. Mit `mktemp()` könnt ihr mehrere temporäre Verzeichnisse erstellen.

In *Test-Fixtures* haben wir die Standardbibliothek `tempfile.TemporaryDirectory` für unser `db`-Fixture verwendet:

```
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()
```

Lasst uns stattdessen eines der neuen Built-ins verwenden. Da unser `db`-Fixture im `session`-Scope liegt, können wir `tmp_path` nicht verwenden, da `session`-Scope-Fixtures keine `function`-Scope-Fixtures verwenden können. Wir können jedoch `tmp_path_factory` verwenden:

```
@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()
```

Bemerkung: Wir können dadurch auch zwei Import-Anweisungen entfernen, da wir weder `pathlib` noch `tempfile` importieren müssen.

Tipp: Verwendet nicht tmpdir oder tmpdir_factory, da diese `py.path.local`-Objekte bereitstellen, ein Legacy-Typ.

Das Basisverzeichnis für alle temporären pytest-Verzeichnisse ist system- und anwendungsabhängig. Es enthält einen `pytest-`*`NUM`*-Teil, wobei *`NUM`* bei jeder Sitzung erhöht wird. Das Basisverzeichnis wird unmittelbar nach einer Sitzung

unverändert belassen, damit ihr es im Falle von Testfehlern untersuchen könnt. pytest räumt sie schließlich auf. Nur die letzten paar temporären Basisverzeichnisse werden auf dem System belassen.

Ihr könnt auch euer eigenes Basisverzeichnis angeben mit `pytest --basetemp=MYDIR`.

capsys

Manchmal soll der Anwendungscode etwas auf `stdout`, `stderr` usw. ausgeben. Das Items-Beispielprojekt hat deswegen auch eine Kommandozeilenschnittstelle, die wir nun testen wollen.

Der Befehl `items version` soll die Version ausgeben:

```
$ items version
0.1.0
```

Die Version ist auch via Python verfügbar:

```
>>> import items
>>> items.__version__
'0.1.0'
```

Eine Möglichkeit, dies zu testen, ist

1. den Befehl mit `subprocess.run()` auszuführen
2. die Ausgabe zu erfassen
3. sie mit der Version aus der API zu vergleichen

```
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__
```

Die Funktion `rstrip()` wird verwendet, um den Zeilenumbruch zu entfernen.

Das capsys-Fixture ermöglicht die Erfassung von Schreibvorgängen auf `stdout` und `stderr`. Wir können die Methode, die dies im CLI implementiert, direkt aufrufen und `capsys` zum Lesen der Ausgabe verwenden:

```
import items


def test_version(capsys):
    items.cli.version()
    output = capsys.readouterr().out.rstrip()
    assert output == items.__version__
```

Die Methode `capsys.readouterr()` gibt ein `namedtuple` zurück, das `out` und `err` enthält. Wir lesen nur den `out`-Teil und entfernen dann den Zeilenumbruch mit `rstrip()`.

Eine weitere Funktion von `capsys` ist die Möglichkeit, die normale Ausgabeerfassung von pytest vorübergehend zu deaktivieren. pytest erfasst normalerweise die Ausgaben eurer Tests und des Anwendungscodes. Dies schließt `print`-Anweisungen ein.

```
import items


def test_stdout():
    version = items.__version__
    print("\nitems " + version)
```

Wenn wir den Test jedoch ausführen, sehen wir keine Ausgabe:

```
$ pytest tests/test_output.py
============================= test session starts ==============================
...
collected 1 item

tests/test_output.py .                                                   [100%]

============================== 1 passed in 0.00s ===============================
```

pytest fängt die gesamte Ausgabe auf. Dies hilft zwar, die Kommandozeilensitzung sauber zu halten, es kann jedoch vorkommen, dass wir die gesamte Ausgabe sehen wollen, auch bei bestandenen Tests. Hierfür können die Option `-s` oder `--capture=no` verwenden:

```
 $ 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 ===============================
```

Eine andere Möglichkeit, die Ausgabe immer einzuschließen, ist `capsys.disabled()`:

```
import items


def test_stdout(capsys):
    with capsys.disabled():
        version = items.__version__
        print("\nitems " + version)
```

Nun wird sie Ausgabe im `with`-Block immer angezeigt, auch ohne die `-s`-Option:

```
$ pytest tests/test_output.py
============================= test session starts ==============================
...
collected 1 item
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
tests/test_output.py
items 0.1.0
.                                                                    [100%]

============================== 1 passed in 0.00s ===============================
```

Siehe auch:

capfd
: Wie `capsys`, erfasst aber die Dateideskriptoren 1 und 2, die normalerweise dasselbe wie `stdout` und `stderr`

capsysbinary
: Während capsys Text erfasst, erfasst capsysbinary Bytes

capfdbinary
: erfasst Bytes in den Dateideskriptoren 1 und 2

caplog
: erfasst Ausgaben, die mit dem Logging-Paket geschrieben wurden

monkeypatch

Mit `capsys` kann ich zwar gut die `stdout` und `stderr`-Ausgabe steuern, aber es ist immer noch nicht die Art, wie ich die CLI testen möchte. Die Items-Anwendung verwendet eine Bibliothek namens Typer, die eine Runner-Funktion enthält um unserem Code so zu testen, wie wir es von einem Befehlszeilentest erwarten würden, der im Prozess bleibt und uns mit Output-Hooks versorgt, z.B.:

```
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__
```

Wir werden diese Methode der Ausgabentests als Ausgangspunkt für die restlichen Tests der Items-CLI verwenden. Ich habe mit den CLI-Tests begonnen, indem ich die Items-Version getestet habe. Um den Rest der CLI zu testen, müssen wir die Datenbank in ein temporäres Verzeichnis umleiten, so wie wir es beim Testen der API unter Verwendung von *Fixtures für Setup und Teardown* getan haben. Hierfür verwenden wir nun monkeypatch:

Ein *Monkey Patch* ist eine dynamische Änderung einer Klasse oder eines Moduls während der Laufzeit. Während des Testens ist *monkey patching* eine bequeme Möglichkeit, einen Teil der Laufzeitumgebung des Anwendungscodes zu übernehmen und entweder Eingabe- oder Ausgabeabhängigkeiten durch Objekte oder Funktionen zu ersetzen, die für das Testen besser geeignet sind. Mit dem eingebauten Fixture `monkeypatch` könnt ihr dies im Kontext eines einzelnen Tests tun. Es wird verwendet, um Objekte, Dicts, Umgebungsvariablen, `PYTHONPATH` oder das aktuelle Verzeichnis zu ändern. Es ist wie eine Mini-Version von *Mocking*. Und wenn der Test endet, wird unabhängig davon, ob er bestanden wurde oder nicht, der ursprüngliche, ungepatchte Code wiederhergestellt und alles rückgängig gemacht, was durch den Patch geändert wurde.

Siehe auch:

How to monkeypatch/mock modules and environments

Das `monkeypatch`-Fixture bietet die folgenden Funktionen:

Funktion	Beschreibung
`setattr(TARGET, NAME, VALUE, raising=True)`[1]	setzt ein Attribut
`delattr(TARGET, NAME, raising=True)`[1]	löscht ein Attribut
`setitem(DICT, NAME, VALUE)`	setzt einen Dict-Eintrag
`delitem(DICT, NAME, raising=True)`[1]	löscht einen Dict-Eintrag
`setenv(NAME, VALUE, prepend=None)`[2]	setzt eine Umgebungsvariable
`delenv(NAME, raising=True)`[1]	löscht eine Umgebungsvariable
`syspath_prepend(PATH)`	erweitert den Pfad `sys.path`
`chdir(PATH)`	wechselt das aktuelle Arbeitsverzeichnis

Wir können `monkeypatch` verwenden, um die CLI auf ein temporäres Verzeichnis für die Datenbank umzuleiten, und zwar auf zweierlei Weise. Beide Methoden erfordern Kenntnisse über den Anwendungscode. Schauen wir uns die Methode `cli.get_path()` in `src/items/cli.py` an:

```
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
```

Diese Methode teilt dem restlichen CLI-Code mit, wo sich die Datenbank befindet. Um uns den Speicherort der Datenbank auf der Kommandozeile ausgeben zu lassen, definieren wir nun auch noch `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
```

Um diese Methoden zu testen, können wir nun entweder die gesamte `get_path()`-Funktion oder das `pathlib.Path()`-Attribut `home` patchen. Hierfür definieren wir in `tests/test_config.py` zunächst eine Hilfsfunktion `run_items_cli`, die dasselbe ausgibt wie `items` auf der Kommandozeile:

```
from typer.testing import CliRunner

import items


```

(Fortsetzung auf der nächsten Seite)

[1] Der `raising`-Parameter teilt pytest mit, ob eine Exception ausgelöst werden soll, wenn das Element (noch) nicht vorhanden ist.

[2] Der `prepend`-Parameter von `setenv()` kann ein Zeichen sein. Wenn er gesetzt ist, wird der Wert der Umgebungsvariablen in *`VALUE`* + `prepend` + *`OLD_VALUE`* geändert.

(Fortsetzung der vorherigen Seite)

```
def run_items_cli(*params):
    runner = CliRunner()
    result = runner.invoke(items.app, params)
    return result.output.rstrip()
```

Anschließend können wir dann unseren Test schreiben, der die gesamte `get_path()`-Funktion patcht:

```
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)
```

Die Funktion `get_path()` aus `items.cli` kann nicht einfach durch `tmp_path` ersetzt werden, da dies ein `pathlib.Path`-Objekt ist, das nicht aufrufbar ist. Daher wird sie durch die `fake_get_path()`-Funktion ersetzt. Alternativ können wir jedoch auch das `home`-Attribut von `pathlib.Path` patchen:

```
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)
```

Monkey patching und *Mocking* verkomplizieren jedoch das Testen, sodass wir nach Möglichkeiten suchen werden, dies zu vermeiden, wann immer es möglich ist. In unserem Fall könnte sinnvoll sein, eine Umgebungsvariable `ITEMS_DB_DIR` zu setzen, die einfach gepatcht werden kann:

```
def test_env_var(monkeypatch, tmp_path):
    monkeypatch.setenv("ITEMS_DB_DIR", str(tmp_path))
    assert run_items_cli("config") == str(tmp_path)
```

Verbleibende Built-in-Fixtures

Built-in-Fixture	Beschreibung
`capfd`, `capfdbinary`, `capsysbinary`	Varianten von `capsys`, die mit Dateideskriptoren und/oder binärer Ausgabe arbeiten.
`caplog`	ähnlich wie `capsys`; wird für Meldungen verwendet, die mit Pythons Logging-System erstellt werden.
`cache`	wird zum Speichern und Abrufen von Werten über mehrere Pytest-Läufe hinweg verwendet. Es erlaubt `last-failed`, `failed-first` und ähnliche Optionen.
`doctest_namespace`	nützlich, wenn ihr pytest verwenden möchtet, um *Doctests* durchzuführen.
`pytestconfig`	wird verwendet, um Zugriff auf Konfigurationswerte, Plugin-Manager und -Hooks zu erhalten.
`record_property`, `record_testsuite_p`	wird verwendet, um dem Test oder der Testsuite zusätzliche Eigenschaften hinzuzufügen. Besonders nützlich für das Hinzufügen von Daten zu einem Bericht, der von CI (Continuous Integration)-Tools verwendet wird.
`recwarn`	wird verwendet, um Warnmeldungen zu testen.
`request`	wird verwendet, um Informationen über die ausgeführte Testfunktion bereitzustellen. wird meist bei der Parametrisierung von Fixtures verwendet
`pytester`, `testdir`	Wird verwendet, um ein temporäres Testverzeichnis bereitzustellen, um die Ausführung und das Testen von pytest-Plugins zu unterstützen. `pytester` ist der `pathlib`-basierte Ersatz für das `py.path`-basierte `testdir`.
`tmpdir`, `tmpdir_factory`	ähnlich wie `tmp_path` und `tmp_path_factory`; dient der Rückgabe eines `py.path.local`-Objekts anstelle eines `pathlib.Path`-Objekts.

Ihr könnt die vollständige Liste der Built-in-Fixtures erhalten, indem ihr `pytest --fixtures` ausführt.

Siehe auch:

- Built-in fixtures

Testparametrisierung

Durch Parametrisierung können wir eine Testfunktion in viele Testfälle umwandeln, um mit weniger Arbeit gründlicher zu testen. Hierfür übergeben wir dem Test mehrere Sätze von Argumenten, um neue Testfälle zu erstellen. Wir werfen einen Blick auf redundanten Code, den wir mit Parametrisierung vermeiden. Dann werden wir uns drei Möglichkeiten ansehen, und zwar in der Reihenfolge, in der sie ausgewählt werden sollten:

- Parametrisierung von Funktionen
- Parametrisierung von Fixtures
- Verwendung einer Hook-Funktion namens `pytest_generate_tests`

Dabei werden wir dasselbe Parametrisierungsproblem mit allen drei Methoden lösen, auch wenn manchmal eine Lösung der anderen vorzuziehen ist.

Testen ohne `parametrize`

Das Senden einiger Werte durch eine Funktion und das Überprüfen der Ausgabe auf Korrektheit ist ein gängiges Muster beim Testen von Software. Der einmalige Aufruf einer Funktion mit einem Satz von Werten reicht jedoch selten aus, um die Funktionen vollständig zu testen. Parametrisiertes Testen ist eine Möglichkeit, mehrere Datensätze durch denselben Test zu schicken und pytest berichten zu lassen, wenn einer der Datensätze fehlschlägt. Um das Problem zu verstehen, das parametrisierte Tests zu lösen versuchen, schreiben wir einige Tests für die API-Methode `finish()` aus `src/items/api.py`:

```
def finish(self, item_id: int):
    """Set an item state to done."""
    self.update_item(item_id, Item(state="done"))
```

Die in der Anwendung verwendeten Zustände sind *todo*, *in progress* und *done*, und `finish()` setzt den Zustand einer Karte auf *done*. Um dies zu testen, könnten wir

1. ein Item-Objekt erstellen und es zur Datenbank hinzufügen, damit wir eine Item haben, mit der wir arbeiten können,
2. `finish()` aufrufen
3. sicherstellen, dass der Endzustand *done* ist.

Eine Variable ist der Startstatus der Karte. Er könnte „todo“, „in progress“ oder sogar schon „done“ sein. Lasst uns alle drei testen:

```
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):
    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"
```

Lassen wir es laufen:

```
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 ===============================
```

Die Testfunktionen sind sehr ähnlich. Die einzigen Unterschiede sind der Ausgangszustand und die Zusammenfassung. Eine Möglichkeit, den redundanten Code zu reduzieren, besteht darin, die drei Funktionen in einer einzigen Funktion zusammenzufassen, etwa so:

```
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"
```

Nun lassen wir `tests/test_finish.py` erneut laufen:

```
$ 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 ===============================
```

Auch dieser Test ist bestanden, und wir haben den überflüssigen Code eliminiert. Aber es ist doch nicht dasselbe:

- Es wird nur ein Testfall gemeldet, statt drei.
- Wenn einer der Testfälle fehlschlägt, wissen wir nicht, welcher es ist, ohne einen Blick auf den Traceback oder andere Debugging-Informationen zu werfen.
- Wenn einer der Testfälle fehlschlägt, werden die darauf folgenden Testfälle nicht ausgeführt. pytest stoppt die Ausführung eines Tests, wenn eine Assertion fehlschlägt.

Funktionen parametrisieren

Um eine Testfunktion zu parametrisieren, fügt der Testdefinition Parameter hinzu und verwendet den `@pytest.mark.parametrize()`-Dekorator, um die an den Test zu übergebenden Argumente zu definieren, etwa so:

```
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"
```

Die `test_finish()`-Funktion hat jetzt ihre ursprüngliche `items_db`-Fixture als Parameter, aber auch zwei neue Parameter: `start_summary` und `start_state`. Diese stimmen direkt mit dem ersten Argument von `@pytest.mark.parametrize()` überein.

1. Das erste Argument von `@pytest.mark.parametrize()` ist eine Liste von Parameter-Namen. Dieses Argument könnte auch eine Liste von Zeichenketten sein, wie z.B. `["start_summary", "start_state"]` oder eine komma-getrennte Zeichenkette `"start_summary, start_state"`.
2. Das zweite Argument von `@pytest.mark.parametrize()` ist unsere Liste von Testfällen. Jedes Element in der Liste ist ein Testfall, der durch ein Tupel oder eine Liste dargestellt wird, die ein Element für jedes Argument enthält, das an die Testfunktion gesendet wird.

pytest führt diesen Test einmal für jedes `(start_summary, start_state)`-Paar durch und meldet jeden als separaten 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 ===============================
```

Diese Verwendung von `parametrize()` funktioniert für unsere Zwecke. Allerdings ist es für diesen Test `start_summary` nicht wirklich wichtig und macht jeden Testfall komplexer. Ändern wir die Parametrisierung in `start_state` und sehen uns an, wie sich die Syntax ändert:

```
import pytest

from items import Item


@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"
```

Wenn wir die Tests jetzt ausführen, konzentrieren sie sich auf die Veränderung, die uns wichtig ist:

```
$ 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 ===============================
```

Die Ausgabe der beiden Beispiele, unterscheidet sich insofern, dass jetzt nur noch der Ausgangszustand aufgelistet wird, also *todo*, *in progress* und *done*. Im vorherigen Beispiel zeigte pytest noch die Werte beider Parameter an, getrennt durch einen Bindestrich -. Wenn sich nur ein Parameter ändert, wird kein Bindestrich benötigt.

Fixtures parametrisieren

Bei der Funktionsparametrisierung rief pytest unsere Testfunktion für jeden Satz von Argumenten, die wir angegeben haben, jeweils einmal auf. Mit der Fixture-Parametrisierung verschieben wir diese Parameter in eine Fixture. pytest ruft die Fixture dann jeweils einmal für jeden Satz von Werten auf, die wir angeben. Anschließend wird jede Testfunktion, die von der Fixture abhängt, für jeden Fixture-Wert einmal aufgerufen. Auch die Syntax ist anders:

```
import pytest

from items import Item


@pytest.fixture(params=["done", "in progress", "todo"])
def start_state(request):
    return request.param
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
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"
```

Das bedeutet, dass pytest `start_state()` dreimal aufruft, jeweils einmal für alle Werte in `params`. Jeder Wert von `params` wird in `request.param` gespeichert, damit das Fixture ihn verwenden kann. Innerhalb von `start_state()` könnten wir Code haben, der vom Parameterwert abhängt. In diesem Fall wird jedoch nur der Wert des Parameters zurückgegeben.

Die Funktion `test_finish()` ist identisch mit der Funktion, die wir bei der Funktionsparametrisierung verwendet haben, jedoch ohne den Dekorator `parametrize`. Da sie `start_state` als Parameter hat, ruft pytest sie einmal für jeden Wert auf, der an die `start_state()`-Fixture übergeben wird. Und nach all dem sieht die Ausgabe genauso aus wie vorher:

```
$ 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 ===============================
```

Auf den ersten Blick erfüllt die Fixture-Parametrisierung in etwa den gleichen Zweck wie die Funktionsparametrisierung, allerdings mit etwas mehr Code. Die Fixture-Parametrisierung hat jedoch den Vorteil, dass für jeden Satz von Argumenten ein Fixture ausgeführt wird. Dies ist nützlich, wenn ihr *Setup* - oder *Teardown*-Code habt, der für jeden Testfall ausgeführt werden muss, z.B. eine andere Datenbankverbindung oder ein anderer Dateiinhalt oder was auch immer.

Es hat auch den Vorteil, dass viele Testfunktionen mit demselben Satz von Parametern ausgeführt werden können. Alle Tests, die die `start_state`-Fixture verwenden, werden alle drei Mal aufgerufen, einmal für jeden Startzustand.

Mit `pytest_generate_tests` parametrisieren

Die dritte Möglichkeit der Parametrisierung ist die Verwendung einer Hook-Funktion namens `pytest_generate_tests`. Hook-Funktionen werden oft von *Plugins* verwendet, um den normalen Arbeitsablauf von pytest zu verändern. Aber wir können viele von ihnen in Testdateien und `conftest.py`-Dateien verwenden.

Die Implementierung des gleichen Ablaufs wie zuvor mit `pytest_generate_tests` sieht wie folgt aus:

```
from items import Item


def pytest_generate_tests(metafunc):
    if "start_state" in metafunc.fixturenames:
        metafunc.parametrize("start_state", ["done", "in progress", "todo"])
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
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"
```

Die `test_finish()`-Funktion hat sich nicht geändert; wir haben nur die Art und Weise geändert, wie pytest den Wert für `initial_state` bei jedem Testaufruf einträgt.

Die `pytest_generate_tests`-Funktion, die wir bereitstellen, wird von pytest aufgerufen, wenn es seine Liste der auszuführenden Tests erstellt. Sie ist sehr leistungsfähig und unser Beispiel ist nur ein einfacher Fall, um die Funktionalität früherer Parametrisierungsmethoden abzugleichen. `pytest_generate_tests` ist jedoch besonders nützlich, wenn wir die Parametrisierungsliste zur Zeit der Testsammlung auf interessante Weise ändern wollen. Hier sind ein paar Möglichkeiten:

- Wir könnten unsere Parametrisierungsliste auf einer Kommandozeilen-Option basierend ändern, die uns `metafunc.config.getoption("--SOME_OPTION")`[1] gibt. Vielleicht fügen wir eine `--excessive`- Option hinzu, um mehr Werte zu testen, oder eine `--quick-Option`, um nur einige wenige zu testen.
- Die Parametrisierungsliste eines Parameters kann auf dem Vorhandensein eines anderen Parameters basieren. Bei Testfunktionen, die zwei zusammenhängende Parameter abfragen, können wir beispielsweise beide mit einem anderen Satz von Werten parametrisieren, als wenn der Test nur einen der Parameter abfragt.
- Wir können zwei verwandte Parameter gleichzeitig parametrisieren zum Beispiel `metafunc.parametrize("TUTORIAL, TOPIC", [("PYTHON BASICS", "TESTING"), ("PYTHON BASICS", "DOCUMENTING"), ("PYTHON FOR DATA SCIENCE, "GIT"), ...])`.

Wir haben nun drei Möglichkeiten der Parametrisierung von Tests kennengelernt. Obwohl wir damit im *finish()*-Beispiel nur drei Testfälle aus einer Testfunktion erstellen, kann die Parametrisierung eine große Anzahl von Testfällen erzeugen.

Markers

Marker in pytest kann man sich wie Tags oder Etiketten vorstellen. Wenn einige Tests langsam sind, könnt ihr sie mit `@pytest.mark.slow` markieren und pytest diese Tests überspringen lassen, wenn ihr in Eile seid. Ihr könnt eine Handvoll Tests aus einer Testsuite auswählen und sie mit `@pytest.mark.smoke` markieren und diese als erste Stufe einer Testpipeline in einem *CI*-System ausführen. Ihr könnt wirklich für jeden Grund, den ihr habt, um nur einige Tests auszuführen, Marker verwenden.

pytest enthält eine Handvoll Built-in-Marker, die das Verhalten der Testausführung verändern. Eine davon, `@pytest.mark.parametrize`, haben wir bereits in *Funktionen parametrisieren* verwendet. Zusätzlich zu den benutzerdefinierten Markierungen, die wir erstellen und zu unseren Tests hinzufügen können, weisen die Built-in-Marker pytest an, etwas Besonderes mit den markierten Tests zu tun.

Im Folgenden werden wir beide Arten von Markern genauer untersuchen: die Built-in-Marker, die das Verhalten ändern, und die benutzerdefinierten Marker, die wir erstellen können, um auszuwählen, welche Tests ausgeführt werden sollen. Wir können Marker auch verwenden, um Informationen an eine Fixture zu übergeben, die von einem Test verwendet wird.

[1] https://docs.pytest.org/en/latest/reference.html#metafunc

Built-in-Markers verwenden

Die Built-in-Markers von pytest werden verwendet, um die Testausführung zu verändern. Hier ist die vollständige Liste der Built-in-Markers, die in pytest enthalten sind:

@pytest.mark.filterwarnings(*WARNUNG*)
: Dieser Marker fügt dem angegebenen Test einen Warnfilter hinzu.

@pytest.mark.skip(reason=*None*)
: Mit diesem Marker wird der Test mit einem optionalen Grund übersprungen.

@pytest.mark.skipif(*BEDINGUNG*, ...*, *GRUND*)
: Diese Markierung überspringt den Test, wenn eine der Bedingungen True ist.

@pytest.mark.xfail(*BEDINGUNG*, ...* *GRUND*, run=*True*, raises=*None*, strict=*xfail_strict*)
: Dieser Marker teilt pytest mit, dass wir das Fehlschlagen erwarten.

@pytest.mark.parametrize({ARG1, ARG2, ...
: Dieser Marker ruft eine Testfunktion mehrfach auf, wobei nacheinander verschiedene Argumente übergeben werden.

@pytest.mark.usefixtures({FIXTURE1, FIXTURE2, ...
: Dieser Marker kennzeichnet Tests, die alle angegebenen Fixtures benötigen.

@pytest.mark.parametrize haben wir bereits verwendet. Lasst uns die drei anderen, am häufigsten verwendeten Built-in-Markers mit einigen Beispielen durchgehen, um zu sehen, wie sie funktionieren.

Überspringen von Tests mit `@pytest.mark.skip`

Der Marker `skip` erlaubt es uns, einen Test zu überspringen. Nehmen wir an, wir wollen in einer zukünftigen Version der Items-Anwendung die Möglichkeit zum Sortieren hinzufügen und möchten, dass die `Item`-Klasse Vergleiche unterstützt. Wir schreiben einen Test für den Vergleich von `Item`-Objekten mit < wie folgt:

```
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
```

Und er scheitert:

```
pytest --tb=short tests/test_compare.py
============================= test session starts ==============================
...
collected 2 items

tests/test_compare.py F.                                                 [100%]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
================================== 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 ==========================
```

Der Fehler liegt einfach daran, dass wir diese Funktion noch nicht implementiert haben. Dennoch müssen wir diesen Test nicht wieder wegwerfen; wir können ihn einfach auslassen:

```
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
```

Der Marker `@pytest.mark.skip()` weist pytest an, den Test zu überspringen. Die Angabe eines Grundes ist zwar optional, aber sie hilft bei der weiteren Entwicklung. Wenn wir übersprungene Tests ausführen, werden sie als `s` angezeigt:

```
$ 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 ==========================
```

... oder verbos als `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 ==========================
```

Da wir pytest mit `-r` angewiesen haben, eine kurze Zusammenfassung unserer Tests auszugeben, erhalten wir eine zusätzliche Zeile am unteren Ende, die den Grund auflistet, den wir im Marker angegeben haben. Das `a` in `-ra` steht

für *all except passed*. Die Optionen `-ra` sind die gebräuchlichsten, da wir fast immer wissen wollen, warum bestimmte Tests nicht bestanden haben.

Siehe auch:

- Skipping test functions

Bedingtes Überspringen von Tests mit `@pytest.mark.skipif`

Angenommen, wir wissen, dass wir die Sortierung in den Versionen 0.1.x der App Items nicht unterstützen werden, wohl aber in Version 0.2.x. Dann können wir pytest anweisen, den Test für alle Versionen von Items, die kleiner als 0.2.x sind, wie folgt zu überspringen:

```
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
```

Mit dem `skipif`-Marker könnt ihr beliebig viele Bedingungen eingeben, und wenn eine davon wahr ist, wird der Test übersprungen. In unserem Fall verwenden wir `packaging.version.parse`, um die Minor-Version zu isolieren und sie mit der Zahl 2 zu vergleichen.

In diesem Beispiel wird als zusätzliches Paket packaging verwendet. Wenn ihr das Beispiel ausprobieren möchtet, installiert es zunächst mit `python -m pip install packaging`.

Tipp: `skipif` ist auch hervorragend geeignet, wenn Tests für verschiedene Betriebssysteme unterschiedlich geschrieben werden müssen.

Siehe auch:

- skipif

`@pytest.mark.xfail`

Wenn wir alle Tests durchführen wollen, auch die, von denen wir wissen, dass sie fehlschlagen werden, können wir den Marker `xfail` oder genauer `@pytest.mark.xfail(`*`CONDITION, ... *, {REASON`*`, run=`*`True`*`, raises=`*`None`*`, strict=`*`True`*`)` verwenden. Der erste Satz von Parametern für diese Fixture ist der gleiche wie bei `skipif`.

run
: Der Test wird standardmäßig ausgeführt, außer wenn `run=False` gesetzt ist.

raises
: erlaubt euch, einen Ausnahmetyp oder ein Tupel von Ausnahmetypen anzugeben, die zu einem `xfail` führen sollen. Jede andere Ausnahme führt dazu, dass der Test fehlschlägt.

strict
: teilt pytest mit, ob bestandene Tests (`strict=False`) als `XPASS` oder mit `strict=True` als `FAIL` markiert werden sollen.

Schauen wir uns ein Beispiel an:

```
import pytest
from packaging.version import parse

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
```

Wir haben hier drei Tests: einen, von dem wir wissen, dass er fehlschlägt, und zwei, von denen wir wissen, dass sie bestanden wird. Diese Tests demonstrieren sowohl das Scheitern als auch das Bestehen der Verwendung von `xfail` und die Auswirkungen der Verwendung von `strict`. Das erste Beispiel verwendet auch den optionalen Parameter `condition`, der wie die Bedingungen von `skipif` funktioniert. Und so sieht das Ergebnis aus:

```
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 ===================================
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
_________________________________ 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, die mit `xfail` gekennzeichnet sind:

- Nicht bestandene Tests werden mit `XFAIL` angezeigt.
- Bestandene Tests mit `strict=False` führen zu `XPASSED`.
- Bestandene Tests mit `strict=True` führen zu `FAILED`.

Wenn ein Test fehlschlägt, der mit `xfail` markiert ist, also mit `XFAIL` ausgegeben wird, hatten wir Recht in der Annahme, dass der Test fehlschlagen wird.

Bei Tests, die mit `xfail` markiert wurden, jedoch tatsächlich bestanden wurden, gibt es zwei Möglichkeiten: Wenn sie zu `XFAIL` führen sollen, dann solltet ihr die Finger von `strict` lassen. Wenn sie hingegen `FAILED` ausgeben sollen, dann setzt `strict`. Ihr könnt `strict` entweder als Option für den `xfail`-Marker setzen, wie wir es in diesem Beispiel getan haben, oder ihr könnt es auch global mit der Einstellung `xfail_strict=True` in der pytest-Konfigurationsdatei `pytest.ini` setzen.

Ein pragmatischer Grund, immer `xfail_strict=True` zu verwenden, ist, dass wir uns alle fehlgeschlagenen Tests üblicherweise genauer anzuschauen. Und so sehen wir uns dann auch die Fälle an, in denen die Erwartungen an den Test nicht mit dem Ergebnis übereinstimmen.

`xfail` kann sehr hilfreich sein wenn ihr in einer testgetriebenen Entwicklung arbeitet und ihr Testfälle schreibt, von denen ihr wisst, dass sie noch nicht implementiert sind, die ihr aber in Kürze implementieren wollt. Lasst dabei die `xfail`-Tests auf dem Feature-Branch, in dem die Funktion implementiert wird.

Oder etwas geht kaputt, ein oder mehrere Test schlagen fehl, und ihr könnt nicht sofort sofort an der Behebung arbeiten. Das Markieren der Tests als `xfail`, `strict=true` mit der Angabe der Fehler-/Issue-Report-ID in `reason`, ist eine gute Möglichkeit, den Test weiterlaufen zu lassen und ihn nicht zu vergessen.

Wenn ihr jedoch nur ein Brainstorming über die Behaviors eurer Anwendung macht, solltet ihr noch keine Tests schreiben und sie mit `xfail` oder `skip` markieren: hier würde ich euch YAGNI ('You Aren't Gonna Need It', deutsch: „Du wirst es nicht brauchen") entgegenhalten. Implementiert Dinge immer erst dann, wenn sie tatsächlich gebraucht werden und niemals, wenn ihr nur ahnt, dass ihr sie brauchen werdet.

Tipp:

- Ihr solltet `xfail_strict = True` in `pytest.ini` setzen, um alle `XPASSED`-Ergebnisse in `FAILED` zu verwandeln.
- Zudem solltet ihr immer `-ra` oder zumindest `-rxX` verwenden um euch den Grund anzeigen zu lassen.
- Und schließlich solltet ihr eine Fehlernummer in `reason` angeben.
- `pytest --runxfail` ignoriert grundsätzlich die `xfail`-Marker. Dies ist sehr nützlich in den letzten Phasen des Pre-Production-Testing.

Auswahl von Tests mit eigenen Markern

Eigene Marker könnt ihr euch wie Tags oder Etiketten vorstellen. Sie können verwendet werden, um Tests auszuwählen, die ausgeführt oder übersprungen werden sollen.

Nehmen wir an, wir wollen einige unserer Tests mit `smoke` kennzeichnen. Die Segmentierung einer Teilmenge von Tests in eine Smoke-Test-Suite ist eine gängige Praxis, um einen repräsentativen Satz von Tests ausführen zu können, der uns schnell sagen kann, ob irgendetwas mit einem der Hauptsysteme nicht in Ordnung ist. Darüber hinaus werden wir einige unserer Tests mit `exception` kennzeichnen – diejenigen, die auf erwartete Ausnahmen prüfen:

```
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)
    s = items_db.get_item(i)
    assert s.state == "in progress"
```

Jetzt sollten wir in der Lage sein, nur diesen Test auszuwählen, indem wir die Option `-m smoke` verwenden:

```
$ 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 ==================
```

Nun konnten wir zwar nur einen Test durchzuführen, aber wir haben auch eine Warnung erhalten: `PytestUnknownMarkWarning: Unknown pytest.mark.smoke - is this a typo?` Sie hilft, Tippfehler zu vermeiden. pytest möchte, dass wir benutzerdefinierte Marker registrieren, indem wir einen Marker-Abschnitt zu `pytest.ini` hinzufügen, z.B.:

```
[pytest]
markers =
    smoke: Small subset of all tests
```

Jetzt warnt uns pytest nicht mehr vor einem unbekannten 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 ========================
```

Machen wir dasselbe mit der `exception`-Markierung für `test_start_non_existent`.

1. Zuerst registrieren wir den Marker in `pytest.ini`:

```
[pytest]
markers =
    smoke: Small subset of tests
    exception: Only run expected exceptions
```

2. Dann fügen wir den Marker zum Test hinzu:

```
@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. Schließlich führen wir den Test mit `-m exception` aus:

```
$ 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 ========================
```

Marker für Dateien, Klassen und Parameter

Mit den Tests in `test_start.py` haben wir `@pytest.mark.MARKER_NAME`-Dekoratoren zu Testfunktionen hinzugefügt. Wir können auch ganze Dateien oder Klassen mit Markern versehen, um mehrere Tests zu markieren, oder in parametrisierte Tests hineingehen und einzelne Parametrisierungen markieren. Wir können sogar mehrere Marker auf einen einzigen Test setzen. Zunächst setzen wir in `test_finish.py` mit einem Marker auf Dateiebene:

```
import pytest

from items import Item

pytestmark = pytest.mark.finish
```

Wenn pytest ein `pytestmark`-Attribut in einem Testmodul sieht, wird es den oder die Marker auf alle Tests in diesem Modul anwenden. Wenn ihr mehr als einen Marker auf die Datei anwenden wollt, könnt ihr eine Listenform verwenden: `pytestmark = [pytest.mark.MARKER_ONE, pytest.mark.MARKER_TWO]`.

Eine andere Möglichkeit, mehrere Tests gleichzeitig zu markieren, besteht darin, Tests in einer Klasse zu haben und Markierungen auf Klassenebene zu verwenden:

```
@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")
        )
        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"
```

Die Testklasse `TestFinish` ist mit `@pytest.mark.smoke` gekennzeichnet. Wenn ihr eine Testklasse auf diese Weise markiert, wird jede Testmethode in der Klasse mit dem gleichen Marker versehen.

Wir können auch nur bestimmte Testfälle eines parametrisierten Tests markieren:

```
@pytest.mark.parametrize(
    "states",
    [
        "todo",
        pytest.param("in progress", marks=pytest.mark.smoke),
        "done",
    ],
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
)
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"
```

Die `test_finish()`-Funktion ist nicht direkt markiert, sondern nur einer ihrer Parameter: `pytest.param("in progress", marks=pytest.mark.smoke)`. Ihr könnt mehr als einen Marker verwenden, indem ihr die Listenform verwendet: `marks=[pytest.mark.ONE, pytest.mark.TWO]`. Wenn ihr alle Testfälle eines parametrisierten Tests markieren wollt, fügt ihr den Marker wie bei einer normalen Funktion entweder über oder unter dem Dekorator `parametrize` ein.

Das vorherige Beispiel bezog sich auf die Funktionsparametrisierung. Ihr könnt jedoch auch Fixtures auf die gleiche Weise markieren:

```
@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"
```

Wenn ihr einer Funktion mehr als eine Markierung hinzufügen wollt, könnt ihr einfach stapeln. Zum Beispiel wird `test_finish_non_existent()` sowohl mit `@pytest.mark.smoke` als auch mit `@pytest.mark.exception` markiert:

```
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)
```

Wir haben in `test_finish.py` eine Reihe von Markern auf verschiedene Weise hinzugefügt. Dabei verwenden wir die Marker, um die auszuführenden Tests anstatt eine Testdatei auszuwählen:

```
$ cd tests
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
$ 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 =======================
```

Marker zusammen mit `and`, `or`, `not` und `()`

Wir können Marker logisch verknüpfen, um Tests auszuwählen, genau wie wir `-k` zusammen mit Schlüsselwörtern zur Auswahl von Testfällen in *Testsuite* verwendet haben. So können wir nur die `finish`-Tests, die sich mit `exception` befassen:

```
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 =======================
```

Wir können auch alle logischen Verknüpfungen zusammen verwenden:

```
 $ 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%]

======================= 2 passed, 34 deselected in 0.08s =======================
```

Schließlich können wir auch Marker und Keywords für die Auswahl kombinieren, z.B. um Smoke-Tests auszuführen, die nicht Teil der Klasse `TestFinish` sind:

```
$ 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%]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
test_start.py::test_start PASSED                                        [100%]

======================== 3 passed, 33 deselected in 0.07s ========================
```

Bei der Verwendung von Markern und Keywords ist zu beachten, dass die Namen der Marker bei der Option `-m` *MARKERNAME* vollständig sein müssen, während Keywords bei der Option `-k` *KEYWORD* eher einen Substring darstellen.

`--strict-markers`

Üblicherweise erhalten wir eine Warnung, wenn ein Marker nicht registriert ist. Wenn diese Warnung stattdessen ein Fehler sein soll, können wir die Option `--strict-markers` verwenden. Dies hat zwei Vorteile:

1. Der Fehler wird bereits ausgegeben, wenn die auszuführenden Tests gesammelt werden und nicht erst zur Laufzeit. Wenn ihr eine Testsuite habt, die länger als ein paar Sekunden dauert, werdet ihr es zu schätzen wissen, wenn ihr diese Rückmeldung schnell erhaltet.
2. Zweitens sind Fehler manchmal leichter zu erkennen als Warnungen, besonders in Systemen mit *kontinuierlicher Integration*.

Tipp: Es empfiehlt sich daher, immer `--strict-markers` zu verwenden. Anstatt die Option jedoch immer wieder einzugeben, könnt ihr `--strict-markers` in den Abschnitt `addopts` der `pytest.ini` einfügen:

```
[pytest]
...
addopts =
    --strict-markers
```

Marker mit Fixtures kombinieren

Marker können in Verbindung mit Fixtures, Plugins und Hook-Funktionen verwendet werden. Die Built-in-Marker benötigen Parameter, während die benutzerdefinierten Marker, die wir bisher verwendet haben, keine Parameter benötigen. Erstellen wir einen neuen Marker namens `num_items`, den wir an die `items_db-Fixture` übergeben können. Die `items_db`-Fixture bereinigt derzeit die Datenbank für jeden Test, der sie verwenden möchte:

```
@pytest.fixture(scope="function")
def items_db(session_items_db):
    db = session_items_db
    db.delete_all()
    return db
```

Wenn wir zum Beispiel vier Items in der Datenbank haben wollen, wenn unser Test beginnt, können wir einfach eine andere, aber ähnliche Fixture schreiben:

```
@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"),
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        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
```

Dann könnten wir die ursprüngliche Fixture für Tests verwenden, die eine leere Datenbank bereitstellt, und die neue Fixture für Tests, die eine Datenbank mit vier Items enthält:

```
def test_zero_item(items_db):
    assert items_db.count() == 0


def test_four_items(populated_db):
    assert populated_db.count() == 4
```

Wir haben nun die Möglichkeit, entweder null oder vier Items in der Datenbank zu testen. Was aber, wenn wir keine, vier oder 13 Items haben wollen? Dann wollen wir nicht jedesmal eine neue Fixture schreiben. Marker erlauben uns, einem Test zu sagen, wieviele Items wir haben wollen. Hierfür sind drei Schritte notwendig:

1. Zunächst definieren wir drei verschiedene Tests in `test_items.py` mit dem unserem 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. Diesen Marker müssen wir dann in der `pytest.ini`-Datei deklarieren:

   ```
   [pytest]
   markers =
       ...
       num_items: Number of items to be pre-filled for the items_db fixture
   ```

3. Nun modifizieren wir die `items_db`-Fixture in der `conftest.py`-Datei, um den Marker verwenden zu können. Um die Item-Informationen nicht hart kodieren zu müssen, werden wir das Python-Paket Faker verwenden, das wir mit `python -m pip install faker` installieren können:

```
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
13 @pytest.fixture(scope="function")
14 def items_db(session_items_db, request, faker):
15     db = session_items_db
16     db.delete_all()
17     # Support for random selection "@pytest.mark.num_items({NUMBER})`.
18     faker.seed_instance(99)
19     m = request.node.get_closest_marker("num_items")
20     if m and len(m.args) > 0:
21         num_items = m.args[0]
22         for _ in range(num_items):
23             db.add_item(
24                 Item(summary=faker.sentence(), owner=faker.first_name())
25             )
26     return db
```

Hier gibt es eine Menge Änderungen, die wir jetzt durchgehen wollen.

Zeile 13
Wir haben `request` und `faker` in die Liste der `items_db`-Parameter aufgenommen.

Zeile 17
Dies setzt die Zufälligkeit von Faker, so dass wir jedes Mal die gleichen Daten erhalten. Dabei verwenden wir Faker hier nicht für sehr zufällige Daten, sondern um zu vermeiden, dass wir selbst Daten erfinden müssen.

Zeile 18
Hier verwenden wir `request`, genauer `request.node` für die pytest-Repräsentation eines Tests. `get_closest_marker('num_items')` gibt ein Marker-Objekt zurück, wenn der Test mit `num_items` markiert ist, andernfalls gibt es `None` zurück. Die `get_closest_marker()`-Funktion gibt den Marker zurück, der dem Test am nächsten liegt, und das ist normalerweise das, was wir wollen.

Zeile 19
Der Ausdruck ist wahr, wenn der Test mit `num_items` markiert ist und ein Argument angegeben wird. Die zusätzliche `len`-Prüfung dient dazu, dass, falls jemand versehentlich nur `pytest.mark.num_items` verwendet, ohne die Anzahl der Items anzugeben, dieser Teil übersprungen wird.

Zeile 20–22
Sobald wir wissen, wie viele Items wir erstellen müssen, lassen wir Faker einige Daten für uns erstellen. Faker stellt die Faker-Fixture zur Verfügung.

- Für das Feld `summary` funktioniert die Methode `faker.sentence()`.
- Für das Feld `Owner` funktioniert die Methode `faker.first_name()`.

Siehe auch:

- Es gibt noch viele andere Möglichkeiten, die ihr mit Faker nutzen könnt. Schaut hierfür in die Faker-Dokumentation.
- Neben Faker gibt es nach weitere Bibliothkeen, die Fake-Daten bereitstellen, siehe *Fake Plugins*.

Führen wir die Tests nun aus, um sicherzustellen, dass alles richtig funktioniert:

```
$ 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 ===============================
```

Bemerkung: Damit ihr einen Eindruck bekommt, wie die Daten von Faker aussehen, könnt ihr eine `print`-Anweisung zu `test_four_items()` hinzufügen:

```
@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)
```

Anschließend könnt ihr die Tests in `test_items.py` erneut aufrufen:

```
$ 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

============================== 3 passed in 0.09s ===============================
```

Marker auflisten

Wir haben bereits eine Menge Marker behandelt: die Built-in-Marker `skip`, `skipif` und `xfail`, unsere eigenen Marker `smoke`, `exception`, `finish` und `num_items` und es gibt auch noch ein paar weitere Built-in-Marker. Und wenn wir anfangen, *Plugins* zu verwenden, können noch weitere Marker hinzukommen. Um alle verfügbaren Marker mit Beschreibungen und Parameter aufzulisten, könnt ihr `pytest --markers` ausführen:

```
$ 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
...
```

Dies ist eine sehr praktische Funktion, mit der wir schnell nach Markern suchen können, und ein guter Grund, nützliche Beschreibungen zu unseren eigenen Markern hinzuzufügen.

Plugins

So leistungsfähig pytest ist, es kann noch mehr, wenn wir Plugins hinzufügen. Die Codebasis von pytest ist so konzipiert, dass Anpassungen und Erweiterungen möglich sind, und es gibt Hooks, die Änderungen und Verbesserungen durch Plugins ermöglichen.

Es wird euch vielleicht überraschen, dass ihr bereits einige Plugins geschrieben habt, wenn ihr die vorherigen Abschnitte durchgearbeitet habt. Jedes Mal, wenn ihr Fixtures oder Hook-Funktionen in die Datei `conftest.py` eines Projekts einfügt, erstellt ihr ein lokales Plugin. Es ist nur ein wenig zusätzliche Arbeit, diese `conftest.py`-Dateien in installierbare Plugins umzuwandeln, die ihr zwischen Projekten, mit anderen Personen oder mit der ganzen Welt teilen könnt.

Zunächst beginnen wir jedoch mit der Frage, wo ihr Plugins von Drittanbietern finden könnt. Es gibt eine ganze Reihe von Plugins, so dass die Wahrscheinlichkeit groß ist, dass Änderung, die ihr an pytest vornehmen wollt, bereits geschrieben wurden.

Plugins finden

Ihr könnt pytest-Plugins von Drittanbietern an verschiedenen Stellen finden, u.a. enthält die pytest-Dokumentation eine alphabetisch geordnete Liste von Plugins, die von *pypi.org* stammen. Ihr könnt auch pypi.org selbst durchsuchen, z.B. nach pytest oder nach dem pytest-Framework. Schließlich finden sich in pytest-dev auf GitHub auch viele beliebte pytest-Plugins.

Plugins installieren

pytest-Plugins lassen sich, wie anderen Python-Pakete einfach mit *pip* installieren: `python -m pip install pytest-cov`.

Plugins für ...

... veränderte Testabläufe

pytest führt unsere Tests üblicherweise in einer vorhersehbaren Reihenfolge aus. Bei einem Verzeichnis mit Testdateien führt pytest jede Datei in alphabetischer Reihenfolge aus. Innerhalb jeder Datei wird jeder Test in der Reihenfolge ausgeführt, in der er in der Datei erscheint. Manchmal kann es jedoch sinnvoll sein, diese Reihenfolge zu ändern. Die folgenden Plugins ändern den üblichen Ablauf eines Test:

pytest-xdist
: führt Tests parallel aus, entweder mit mehreren CPUs auf einer Maschine oder mehreren entfernten Maschinen.

pytest-rerunfailures
: führt fehlgeschlagene Tests erneut aus und ist v.a. (vor allem) hilfreich bei fehlerhaften Tests.

pytest-repeat
: macht es einfach, einen oder mehrere Tests zu wiederholen.

pytest-order
: ermöglicht die Festlegung der Reihenfolge durch *Markers*.

pytest-randomly
: lässt die Tests in zufälliger Reihenfolge ablaufen, zuerst nach Datei, dann nach Klasse, dann schließlich nach Testdatei.

... veränderten Output

Die normale pytest-Ausgabe zeigt hauptsächlich Punkte für bestandene Tests und Zeichen für andere Ausgaben. Wenn ihr `-v` übergebt, seht ihr eine Liste von Testnamen mit dem Ergebnis. Es gibt jedoch Plugins, die die Ausgabe noch weiter verändern:

pytest-instafail
: fügt eine `--instafail`-Option hinzu, das Tracebacks und Ausgaben von fehlgeschlagenen Tests direkt nach dem Fehlschlag meldet. Normalerweise meldet pytest Tracebacks und Ausgaben von fehlgeschlagenen Tests erst, nachdem alle Tests abgeschlossen wurden.

pytest-sugar
: zeigt grüne Häkchen anstelle von Punkten für bestandene Tests und hat einen schönen Fortschrittsbalken. Es zeigt, wie pytest-instafail auch, Fehlschläge sofort an.

pytest-html
: ermöglicht die Erstellung von HTML-Berichten. Berichte können mit zusätzlichen Daten und Bildern, wie z.B. Screenshots von Fehlerfällen, erweitert werden.

pytest-icdiff
: verbessert Diffs in den Fehlermeldungen der Pytest-Assertion mit ICDiff.

... für die Webentwicklung

pytest wird ausgiebig für das Testen von Webprojekten verwendet und es gibt eine lange Liste von Plugins, die das Testen weiter vereinfachen:

pytest-selenium
: stellt Fixtures zur Verfügung, die eine einfache Konfiguration von browserbasierten Tests mit Selenium ermöglichen.

pytest-splinter
: bieten die High-Level-API des auf Selenium aufbauenden Splinter um einfacher von pytest aus verwendet zu werden.

pytest-httpx
: erleichtert das Testen von HTTPX und FastAPI-Anwendungen.

... für Fake-Daten

Wir haben Faker schon verwendet in *Marker mit Fixtures kombinieren*, um mehrere Item-Instanzen zu erzeugen. Es gibt viele Fälle in verschiedenen Bereichen, in denen es hilfreich ist, Fake-Daten zu erzeugen. Es überrascht daher nicht, dass es mehrere Plugins gibt, die diesen Bedarf decken:

Faker
: generiert Fake-Daten für euch und bietet ein Faker Fixture für die Verwendung mit pytest.

pytest-factoryboy
: enthält Fixtures für factory-boy, einen Datenbankmodelldatengenerator.

pytest-mimesis
: erzeugt Fake-Daten ähnlich wie Faker, aber Mimesis ist um einiges schneller.

... für Verschiedenes

pytest-cov
: führt die *Coverage* beim Testen aus.

pytest-benchmark
: führt Benchmark-Timing für Code innerhalb von Tests durch.

pytest-timeout
: lässt Tests nicht zu lange laufen.

pytest-asyncio
: testet asynchrone Funktionen.

pytest-mock
: ist ein dünner Wrapper um die *unittest.mock*-Patching-API.

pytest-freezegun
: friert die Zeit ein, so dass jeder Code, der die Zeit, Datum oder Uhrzeit, liest, während eines Tests denselben Wert erhält.

pytest-grpc
: ist ein Pytest-Plugin für gRPC.

pytest-bdd
: schreibt BDD (Behavior Driven Development, deutsch: verhaltensgetriebene Softwareentwicklung)-Tests mit pytest.

Eigene Plugins

Siehe auch:

- Writing plugins

Konfiguration

Mit Konfigurationsdateien könnt ihr den Ablauf von pytest beeinflussen. Wenn ihr immer wieder bestimmte Optionen in euren Tests verwendet, wie `--verbose` oder `--strict-markers`, könnt ihr diese in einer Konfigurationsdatei ablegen und müsst sie nicht immer wieder eingeben. Zusätzlich zu den Konfigurationsdateien gibt es eine Handvoll anderer Dateien, die bei der Verwendung von pytest nützlich sind, um die Arbeit beim Schreiben und Ausführen von Tests zu erleichtern:

`pytest.ini`
: Dies ist die wichtigste Konfigurationsdatei von pytest, mit der ihr das Standardverhalten von pytest ändern könnt. Sie legt auch das Stammverzeichnis von pytest fest, oder `rootdir`.

`conftest.py`
: Diese Datei enthält *Test-Fixtures* und Hook-Funktionen. Sie kann in `rootdir` oder in einem beliebigen Unterverzeichnis existieren.

`__init__.py`
: Wenn diese Datei in Test-Unterverzeichnissen abgelegt wird, ermöglicht sie die Verwendung identischer Testdateinamen in mehreren Testverzeichnissen.

Wenn ihr bereits eine `tox.ini`, `pyproject.toml` oder `setup.cfg` in eurem Projekt habt, können sie an die Stelle der `pytest.ini`-Datei treten: `tox.ini` wird von *tox* verwendet, `pyproject.toml` und `setup.cfg` werden für die Paketierung von Python-Projekten verwendet und können zum Speichern von Einstellungen für verschiedene Werkzeuge, einschließlich pytest, verwendet werden.

Ihr solltet eine Konfigurationsdatei haben, entweder `pytest.ini`, oder einem `pytest`-Abschnitt in `tox.ini`, `pyproject.toml` oder in `setup.cfg`.

Sie Konfigurationsdatei legt das oberste Verzeichnis fest, von dem aus `pytest` gestartet wird.

Schauen wir uns einige dieser Dateien im Zusammenhang mit einer Projektverzeichnisstruktur an:

```
items
├── ...
├── pytest.ini
├── src
│   └── ...
└── tests
    ├── __init__.py
    ├── conftest.py
    └── test_....py
```

Im Falle des `items`-Projekts, das wir bisher zum Testen verwendet haben, gibt es auf der obersten Ebene eine `pytest.ini`-Datei und ein Verzeichnis `tests`. Wir werden uns auf diese Struktur beziehen, wenn wir im weiteren Verlauf dieses Abschnitts über die verschiedenen Dateien sprechen.

Speichern von Einstellungen und Optionen 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]` kennzeichnet den Beginn des pytest-Abschnitts. Danach folgen die einzelnen Einstellungen. Bei Konfigurationseinstellungen, die mehr als einen Wert zulassen, können die Werte entweder in eine oder in mehrere Zeilen geschrieben werden in der Form *`EINSTELLUNG = WERT1 WERT2`*. Bei `markers` hingegen ist nur ein Marker pro Zeile erlaubt.

Dieses Beispiel ist eine einfache `pytest.ini`-Datei, die ich so, oder so ähnlich in fast allen meinen Projekten verwende. Gehen wir kurz die einzelnen Zeilen durch:

`addopts =`
: erlaubt die Angabe der pytest-Optionen, die wir immer in diesem Projekt ausführen wollen.

`--strict-markers`
: weist pytest an, bei jedem nicht registrierten Marker, der im Testcode auftaucht, einen Fehler statt einer Warnung auszugeben. Hierdurch können wir Tippfehler bei Marker-Namen vermeiden.

`--strict-config`
: weist pytest an, wenn beim Parsen von Konfigurationsdateien Schwierigkeiten auftauchen, nicht nur eine Warnung sondern einen Fehler auszugeben. Damit vermeiden wir, dass Tippfehler in der Konfigurationsdatei unbemerkt bleiben.

`-ra`
: weist pytest an, am Ende eines Testlaufs nicht nur zusätzliche Informationen zu *Failures* und *Errors* anzuzeigen sondern auch eine Testzusammenfassung.

  `-r`
  : zeigt zusätzliche Informationen zur Testzusammenfassung an.

  `a`
  : zeigt alle außer den bestanden Tests an. Dies fügt den *Failures* und *Errors* die Informationen `skipped`, `xfailed` oder `xpassed` hinzu.

`testpaths = tests`
: teilt pytest mit, wo es nach Tests suchen soll, wenn ihr auf der Kommandozeile keinen Datei- oder Verzeichnisnamen angegeben habt. In unserem Fall sucht pytest im `tests`-Verzeichnis.

  Auf den ersten Blick mag es überflüssig erscheinen, `testpaths` auf `tests` zu setzen, da pytest sowieso dort sucht, und wir keine `test_`-Dateien in unseren `src`- oder `docs`-Verzeichnissen haben. Allerdings kann die Angabe eines `testpaths`-Verzeichnisses ein wenig Startzeit sparen, besonders wenn unsere `src`- oder `docs`- oder andere Verzeichnisse recht groß sind.

`markers =`
: wird verwendet, um Marker zu deklarieren, wie in *Auswahl von Tests mit eigenen Markern* beschrieben.

Siehe auch:

In den Konfigurationsdateien könnt ihr viele weitere Konfigurationseinstellungen und Befehlszeilenoptionen angeben, die ihr euch mit dem Befehl `pytest --help` anzeigen lassen könnt.

Andere Konfigurationsdateien verwenden

Wenn ihr Tests für ein Projekt schreibt, das bereits eine `pyproject.toml`, `tox.ini` oder `setup.cfg`-Datei hat, könnt ihr `pytest.ini` verwenden, um eure pytest-Konfigurationseinstellungen zu speichern, oder ihr könnt eure Konfigurationseinstellungen in einer dieser alternativen Konfigurationsdateien speichern. Die Syntax der beiden Nicht-ini-Dateien unterscheidet sich ein wenig, daher werden wir uns beide Dateien genauer ansehen.

pyproject.toml

Die `pyproject.toml`-Datei war ursprünglich für die Paketierung von Python-Projekten gedacht; sie kann jedoch auch für die Definition von Projekteinstellungen verwendet werden.

Da TOML ein anderer Standard für Konfigurationsdateien ist als `.ini`-Dateien, ist das Format auch ein wenig anders:

```
[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"
    ]
```

Anstelle von `[pytest]` beginnt der Abschnitt mit `[tool.pytest.ini_options]`, die Werte müssen in Anführungszeichen gesetzt werden und Listen von Werten müssen Listen von Zeichenketten in eckigen Klammern sein.

setup.cfg

Das Dateiformat der `setup.cfg` entspricht einer `.ini`-Datei:

```
[tool:pytest]
addopts =
    --strict-markers
    --strict-config
    -ra
testpaths = tests
markers =
    smoke: Small subset of all tests
    exception: Only run expected exceptions
```

Der einzige Unterschied zwischen dieser und der `pytest.ini` ist die Angabe des Abschnitts `[tool:pytest]`.

> **Warnung:** Der Parser der `.cfg`-Datei unterscheidet sich jedoch vom Parser der `.ini`-Datei, und dieser Unterschied kann Probleme verursachen, die schwer aufzuspüren sind, s.a. pytest-Dokumentation.

`rootdir` festlegen

Noch bevor pytest nach auszuführenden Testdateien sucht, liest es die Konfigurationsdatei `pytest.ini`, `tox.ini`, `pyproject.toml` oder `setup.cfg`, die einen pytest-Abschnitt enthält:

- wenn ihr ein Testverzeichnis angegeben habt, beginnt pytest dort zu suchen
- wenn ihr mehrere Dateien oder Verzeichnisse angegeben habt, beginnt pytest mit dem übergeordneten Verzeichnis
- wenn ihr keine Datei oder kein Verzeichnis angebt, beginnt pytest im aktuellen Verzeichnis.

Wenn pytest eine Konfigurationsdatei im Startverzeichnis findet, ist das die Wurzel und wenn nicht, geht pytest den Verzeichnisbaum hoch, bis es eine Konfigurationsdatei findet, die einen pytest-Abschnitt enthält. Sobald pytest eine Konfigurationsdatei gefunden hat, markiert es das Verzeichnis, in dem es sie gefunden hat, als `rootdir`. Dieses Wurzelverzeichnis ist auch die relative Wurzel der IDs. pytest sagt euch auch, wo es eine Konfigurationsdatei gefunden hat. Durch diese Regeln können wir Tests auf verschiedenen Ebenen durchführen und sicher sein, dass pytest die richtige Konfigurationsdatei findet:

```
$ 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` für die gemeinsame Nutzung von lokalen Fixtures und Hook-Funktionen

Die `conftest.py`-Datei wird verwendet, um Fixtures und Hook-Funktionen zu speichern, s.a. *Test-Fixtures* und *Plugins*. Ihr könnt so viele `conftest.py`-Dateien in einem Projekt haben, wie ihr wollt. Alles, was in einer `conftest.py`-Datei definiert ist, gilt für Tests in diesem Verzeichnis und allen Unterverzeichnissen. Wenn ihr eine `conftest.py`-Datei auf der obersten Testebene habt, können die dort definierten Fixtures für alle Tests verwendet werden. Wenn es dann spezielle Fixtures gibt, die nur für ein Unterverzeichnis gelten, können diese in einer anderen `conftest.py`-Datei in diesem Unterverzeichnis definiert werden. Zum Beispiel könnten die CLI-Tests andere Fixtures benötigen als die API-Tests, und einige könnt ihr auch gemeinsam nutzen.

Tipp: Es ist jedoch eine gute Idee, nur eine einzige `conftest.py`-Datei zu halten, damit ihr die Fixture-Definitionen leicht finden können. Auch wenn wir mit `pytest --fixtures -v` immer herausfinden können, wo eine Fixture definiert ist, so ist es dennoch einfacher, wenn sie immer in der einen `conftest.py`-Datei definiert ist.

`__init__.py` um Kollision von Testdateinamen zu vermeiden

Die `__init__.py`-Datei erlaubt es, doppelte Testdateinamen zu haben. Wenn ihr `__init__.py`-Dateien in jedem Test-Unterverzeichnis habt, könnt ihr denselben Testdateinamen in mehreren Verzeichnissen verwenden, z.B.:

```
items
├── ...
├── pytest.ini
├── src
│   └── ...
└── tests
    ├── api
    │   ├── __init__.py
    │   └── test_add.py
    ├── cli
    │   ├── __init__.py
    │   ├── conftest.py
    │   └── test_add.py
    └── conftest.py
```

Nun können wir die `add`-Funktionalität sowohl über die API als auch über die CLI testen, wobei eine `test_add.py` in beiden Verzeichnissen liegt:

```
$ 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 ===============================
```

Die meisten meiner Projekte starten mit folgender Konfiguration:

```
addopts =
   --strict-markers
   --strict-config
   -ra
```

Siehe auch:

- Configuration
- Configuration Options

Debugging von Testfehlern

Wenn Tests fehlschlagen, müssen wir herausfinden, warum. Vielleicht liegt es am Test, vielleicht aber auch an der Anwendung. Der Prozess, um herauszufinden, wo das Problem liegt und was man dagegen tun kann, ist ähnlich.

pytest bietet viele Werkzeuge, die uns helfen können, ein Problem schneller zu lösen, ohne dass wir zu einem Debugger greifen müssen. Python enthält einen eingebauten Quellcode-Debugger namens `pdb`, sowie mehrere Optionen, die das Debuggen mit `pdb` schnell und einfach machen.

Im Folgenden werden wir einige fehlerhafte Codes mit Hilfe von pytest-Optionen und `pdb` debuggen und uns dabei die Debugging-Optionen und die Integration von pytest und `pdb` anzusehen.

Debuggen mit pytest-Optionen

pytest enthält eine ganze Reihe von Kommandozeilen-Optionen, die für die Fehlersuche nützlich sind. Wir werden einige davon verwenden, um unsere Testfehler zu beheben. Optionen für die Auswahl, welche Tests in welcher Reihenfolge ausgeführt werden sollen und wann sie gestoppt werden sollen.

In all diesen Beschreibungen bezieht sich der Begriff *Fehler* auf eine fehlgeschlagene *Assertion* oder eine andere nicht abgefangene *Exception*, die in unserem Quell- oder Testcode, einschließlich der Fixtures, gefunden wurde.

1. Erneute Ausführung fehlgeschlagener Tests

   Beginnen wir mit der Fehlersuche, indem wir sicherstellen, dass die Tests fehlschlagen, wenn wir sie erneut ausführen. Hierfür verwenden wir `--lf`, um nur die fehlgeschlagenen Tests erneut auszuführen, und `--tb=no`, um den Traceback auszublenden. So wissen wir , dass wir den Fehler reproduzieren können.

   1. Nun können wir mit dem Debuggen des ersten Fehlers beginnen und führen hierzu den ersten fehlgeschlagenen Testaus, halten nach dem Fehler an und sehen uns den Traceback an: `pytest --lf -x`.
   2. Um sicher zu gehen, dass wir das Problem verstehen, können wir den gleichen Test mit `-l`/`--showlocals` noch einmal durchführen. Wir brauchen den vollständigen Traceback nicht noch einmal, also können wir ihn mit `--tb=short` kürzen: `pytest --lf -x -l --tb=short`.

      `-l`/`--showlocals` sind oft sehr hilfreich und manchmal gut genug, um einen Testfehler vollständig zu erkennen.

2. Fehlersuche mit pdb

   PDB (Python Debugger) ist Teil der Python Standardbibliothek, so dass wir nichts installieren müssen, um es zu benutzen. Ihr könnt pdb von pytest aus auf verschiedene Weise starten:

   - Fügt einen `breakpoint()`-Aufruf entweder zum Test- oder zum Anwendungscode hinzu. Wenn ein pytest Lauf auf einen `breakpoint()`-Funktionsaufruf trifft, wird er dort anhalten und pdb starten.
   - Verwendet die `--pdb`-Option. Mit `--pdb` hält pytest an der Stelle des Fehlers an.
   - Verwendet die Kombination der `--lf` und `--trace`-Optionen. Mit `--trace` hält pytest am Anfang eines jeden Tests.

     Nachfolgend sind die üblichen Befehle aufgeführt, die von pdb erkannt werden:

Optionen	Beschreibung
Meta-Befehle	
`h(elp)`	gibt eine Liste von Befehlen aus.
`h(elp)` *COMMAND*	gibt die Hilfe zu einem Befehl aus.
`q(uit)`	beendet pdb.
Sehen, wo ihr seid	
`l(ist)`	listet elf Zeilen um die aktuelle Zeile auf; beim erneuten Aufruf werden die nächsten elf Zeilen aufgelistet.
`l(ist) .`	Das Gleiche wie oben, aber mit einem Punkt. Listet elf Zeilen um die aktuelle Zeile auf. Praktisch, wenn ihr `l(ist)` ein paar Mal benutzt habt und eure aktuelle Position verloren habt.
`l(ist)` `first|las`	listet eine bestimmte Gruppe von Zeilen auf.
`ll`	listet den gesamten Quellcode für die aktuelle Funktion auf.
`w(here)`	gibt den Stack-Trace aus.
Werte ansehen	
`p(rint)` *EXPR*	wertet *EXPR* aus und gibt Wert aus.
`pp` *EXPR*	entspricht `p(rint)` *EXPR*, verwendet aber `pretty-print` aus dem pprint-Modul.
`a(rgs)`	gibt die Argumentliste der aktuellen Funktion aus.
Ausführungsbefehle	
`s(tep)`	führt die aktuelle Zeile aus und springt zur nächsten Zeile in Ihrem Quellcode, auch wenn sie sich innerhalb einer Funktion befindet.
`n(ext)`	führt die aktuelle Zeile aus und springt zur nächsten Zeile in der aktuellen Funktion.
`c(ontinue)`	wird bis zum nächsten Haltepunkt fortgesetzt. Bei Verwendung mit `--trace` bis zum Beginn des nächsten Tests fortgesetzt.
`unt(il)` *LINENO*	wird bis zur angegebenen Zeilennummer fortgesetzt.

Siehe auch:

Die vollständige Liste findet ihr in Debugger Commands der pdb-Dokumentation.

Kombinieren von pdb und tox

Um pdb mit tox kombinieren zu können, müssen wir sicherstellen, dass wir Argumente durch tox an pytest übergeben können. Dies geschieht mit der `{posargs}`-Funktion von tox, die in *pytest-Parameter an tox übergeben* beschrieben wurde. Wir haben diese Funktion bereits in unserer `tox.ini` für Items eingerichtet:

```
[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}
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
[gh-actions]
python =
  3.8: py38
  3.9: py39
  3.10: py310
  3.11: py311
```

Wir möchten die Python 3.11-Umgebung ausführen und den Debugger bei einem fehlgeschlagenen Test starten mit `tox -e py311 -- --pdb --no-cov`. Das bringt uns in den pdb, genau an der *Assertion*, die fehlgeschlagen ist.

Nachdem wir den Fehler gefunden und behoben haben, können wir die Tox-Umgebung mit diesem einen Testfehler erneut ausführen: `tox -e py311 -- --lf --tb=no --no-cov`.

Überblick über die gebräuchlichsten pytest-Debugger-Optionen

Optionen	Beschreibung
Optionen für die Auswahl, welche Tests in welcher Reihenfolge ausgeführt werden sollen und wann sie gestoppt werden sollen:	
`--lf`, `--last-failedlf`	führt den zuerst fehlgeschlagenen Test aus
`--ff`, `--failed-first`	startet mit dem zuerst fehlgeschlagenen Test und führt dann alle aus.
`-x`, `--exitfirst`	hält beim ersten Fehler an und führt dann alle aus.
`-maxfail=`*`NUM`*	stoppt die Tests nach *`NUM`* Fehlern.
`--nf`, `--new-first`	führt zuerst neue Testdateien aus, dann den Rest sortiert nach Änderungsdatum.
`--sw`, `--stepwise`	führt den letzten fehlgeschlagenen Test aus, stoppt dann beim nächsten Fehler und startet beim nächsten Mal wieder beim letzten fehlgeschlagenen Test. Ähnlich wie die Kombination von `--lf -x`, aber effizienter.
`--sw-skip`, `--stepwise-skip`	wie oben, aber ein fehlgeschlagener Test wird übersprungen.
Optionen zur Kontrolle der pytest-Ausgabe:	
`-v`, `--verbose:`	verbos, `-vv` ist noch ausführlicher
`--tb`	Traceback-Stil: `[auto|long|short|line|native|no]` Üblicherweise nutze ich `--tb=short` als Standardeinstellung in der Konfigurationsdatei und die anderen für die Fehlersuche.
`-l`, `--showlocals`	zeigt lokale Variablen neben dem Stacktrace an.
Optionen zum Starten eines Kommandozeilen-Debuggers:	
`--pdb`	startet den Python-Debugger im Fehlerfall. Sehr nützlich zum Debuggen mit *tox*.
`--trace`	startet den pdb-Quellcode-Debugger sofort bei der Ausführung jedes Tests.
`--pdbcls`	verwendet Alternativen zu pdb, z.B. den IPython-Debugger mit `--pdb-cls = IPython.terminal.debugger:TerminalPdb`

16.6 Coverage

Wir haben eine erste Liste von Testfällen erstellt. Die Tests im `tests/api`-Verzeichnis testen die Items über die API. Aber woher wissen wir, ob diese Tests unseren Code umfassend testen? An dieser Stelle kommt die Codeabdeckung (engl.: Coverage) ins Spiel.

Tools, die die Codeabdeckung messen, beobachten euren Code, während eine Testsuite ausgeführt wird, und halten fest, welche Zeilen durchlaufen werden und welche nicht. Dieses Maß – die sog. line coverage – wird berechnet, indem die Gesamtzahl der ausgeführten Zeilen durch die Gesamtanzahl der Codezeilen geteilt wird. Code-Coverage-Tools können euch auch sagen, ob alle Pfade in Control-Statements durchlaufen werden, eine Messung, die als Branch-Coverage bezeichnet wird.

Die Codeabdeckung kann euch jedoch nicht sagen, ob eure Testsuite gut ist; sie kann euch nur darüber informieren, wie viel des Anwendungscodes von eurer Testsuite durchlaufen wird.

Coverage.py ist das bevorzugte Python-Tool, das die Codeabdeckung misst. Und pytest-cov ist ein beliebtes *Pytest-Plugin*, das oft in Verbindung mit Coverage.py verwendet wird.

16.6.1 Coverage.py mit pytest-cov verwenden

Sowohl Coverage.py als auch pytest-cov sind Third-Party-Packages, die vor der Verwendung installiert werden müssen:

Ihr könnt einen Report für die Testabdeckung erstellen mit Coverage.py.

```
$ bin/python -m pip install coverage pytest-cov
```

```
C:> Scripts\python -m pip install coverage pytest-cov
```

Bemerkung: Wollt ihr die Testabdeckung für Python 2 and Python<3.6 ermitteln, müsst ihr Coverage<6.0 verwenden.

Um Tests mit Coverage.py auszuführen, müsst ihr die Option `--cov` hinzufügen und entweder einen Pfad zu dem Code angeben, den ihr messen wollt, oder das installierte Paket, das ihr testet. In unserem Fall ist das Projekt Items ein installiertes Paket, so dass wir es mit `--cov=items` testen werden.

Auf die normale pytest-Ausgabe folgt der Abdeckungsbericht, wie hier gezeigt:

```
$ 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%]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
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%
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 ==============================
```

Die vorherige Ausgabe wurde von den Berichtsfunktionen von coverage erzeugt, obwohl wir coverage nicht direkt aufgerufen haben. `pytest --cov=items` wies das `pytest-cov`-Plugin an

- `coverage` mit `--source` auf `items` zu setzen, während pytest mit den Tests ausgeführt wird
- `coverage report` auszuführen für den Line-Coverage-Report

Ohne pytest-cov würden die Befehle wie folgt aussehen:

```
$ coverage run --source=items -m pytest
$ coverage report
```

Die Dateien `__init__.py` und `db.py` haben eine Abdeckung von 100%, was bedeutet, dass unsere Testsuite auf jede Zeile in diesen Dateien trifft. Das sagt uns jedoch nicht, dass sie ausreichend getestet ist oder dass die Tests mögliche Fehler erkennen; aber es sagt uns zumindest, dass jede Zeile während der Testsuite ausgeführt wurde.

Die Datei `cli.py` hat eine Abdeckung von 76%. Dies mag überraschend hoch erscheinen, da wir die CLI noch gar nicht getestet haben. Dies hängt jedoch damit zusammen, dass `cli.py` von `__init__.py` importiert wird, so dass alle Funktionsdefinitionen ausgeführt werden, aber keiner der Funktionsinhalte.

Wirklich interessiert uns jedoch die `api.py`-Datei mit 99% Testabdeckung. Wir können herausfinden, was übersehen wurde, indem wir die Tests erneut ausführen und die Option `--cov-report=term-missing` hinzufügen:

```
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%]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
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
-----------------------------------------------------
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 ==============================
```

Da wir nun die Zeilennummern der nicht getesteten Zeilen haben, können wir die Dateien in einem Editor öffnen und die fehlenden Zeilen betrachten. Einfacher ist es jedoch, sich den HTML-Bericht anzusehen.

Siehe auch:

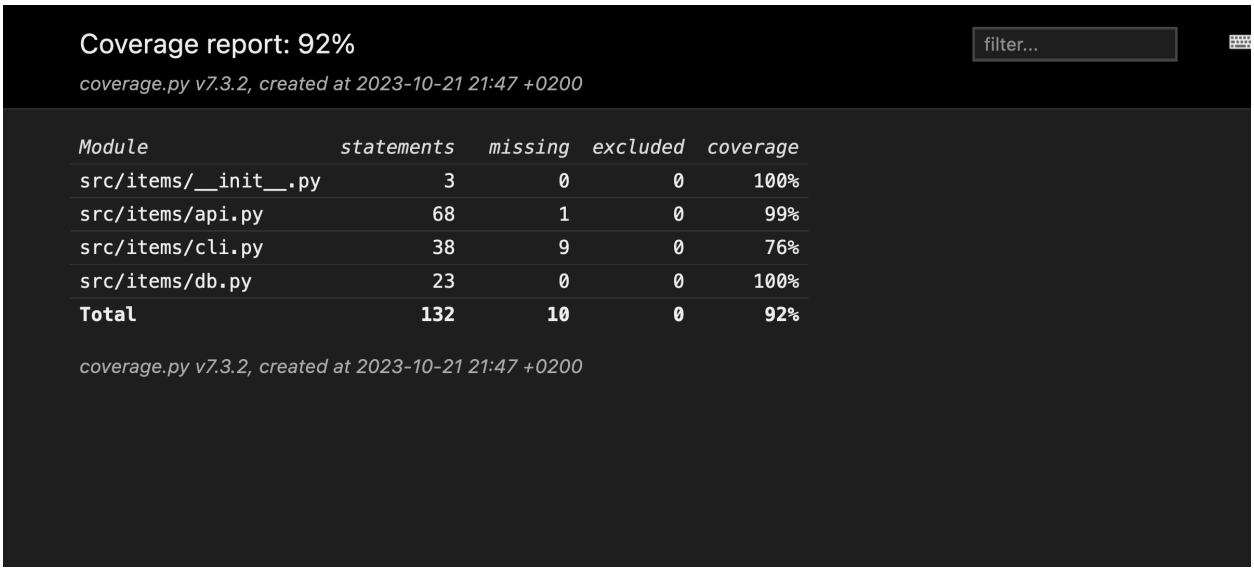
- pytest-cov's documentation

HTML-Berichte generieren

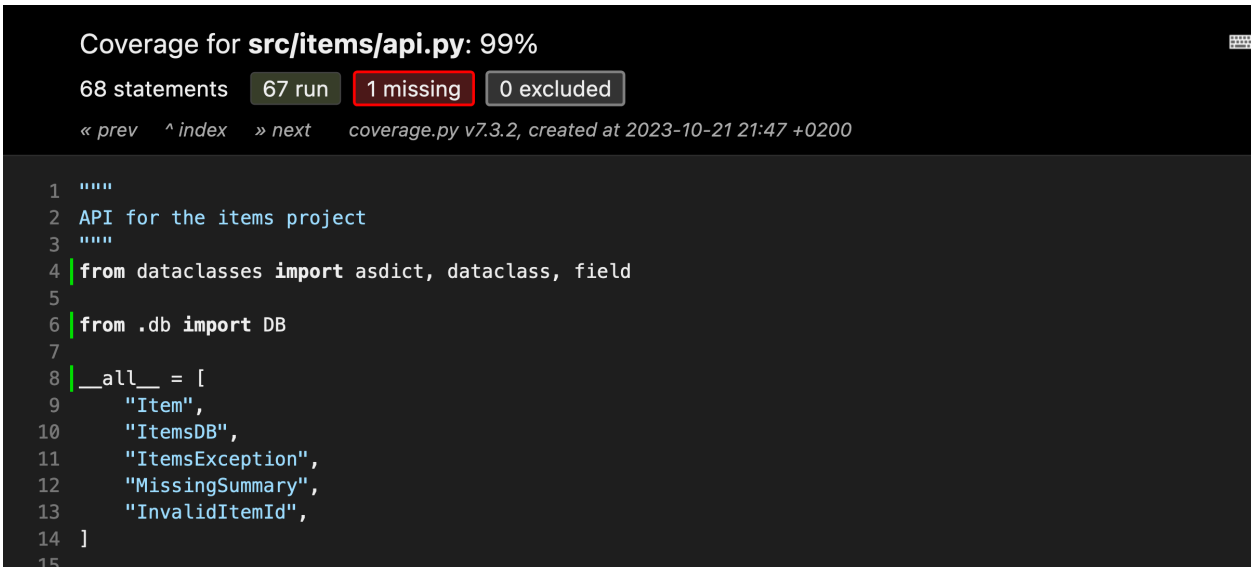
Mit Coverage.py können wir HTML-Berichte erstellen, um die Coverage-Daten detaillierter betrachten zu können. Der Bericht wird entweder mit der Option `--cov-report=html` oder durch die Ausführung von `coverage html` nach einem vorherigen Coverage-Run erstellt:

```
$ cd /PATH/TO/items
$ python3 -m venv .
$ . bin/acitvate
$ python -m pip install ".[dev]"
$ pytest --cov=items --cov-report=html
```

Bei beiden Befehlen wird Coverage.py aufgefordert, einen HTML-Bericht im `htmlcov/`-Verzeichnis zu erstellen. Öffnet `htmlcov/index.html` mit einem Browser und ihr solltet folgendes sehen:



Wenn ihr auf die `src/items/api.py:`-Datei klickt, wird ein Bericht für diese Datei angezeigt:



Der obere Teil des Berichts zeigt den Prozentsatz der abgedeckten Zeilen (99%), die Gesamtzahl der Statements (68) und wie viele Statements ausgeführt (67), übersehen (1) und ausgeschlossen (0) wurden. Klickt auf *missing* , um die Zeilen hervorzuheben, die nicht ausgeführt wurden:

```
src/items/api.py: 99%  67  1  0
49      def add_item(self, item: Item):
50          """Add an item, return the id of the item."""
51          if not item.summary:
52              raise MissingSummary
53          if item.owner is None:
54              item.owner = ""
55          item_id = self._db.create(item.to_dict())
56          self._db.update(item_id, {"id": item_id})
57          return item_id
58
59      def get_item(self, item_id: int):
60          """Return an item with a corresponding id."""
61          db_item = self._db.read(item_id)
62          if db_item is not None:
63              return Item.from_dict(db_item)
64          else:
65              raise InvalidItemId(item_id)
66
67      def list_items(self, owner=None, state=None):
```

Es sieht so aus, als hätte die Funktion `add_item()` eine Exception `MissingSummary`, die bisher nicht getestet wird.

Code von der Testabdeckung ausschließen

In den HTML-Berichten findet ihr eine Spalte mit der Angabe *0 excluded*. Dies bezieht sich auf eine Funktion von Coverage.py, die es uns ermöglicht, einige Zeilen von der Prüfung auszuschließen. In Items schließen wir nichts aus. Es ist jedoch nicht ungewöhnlich, dass einige Codezeilen von der Berechnung der Testabdeckung ausgeschlossen werden, z.B. können Module, die sowohl importiert wie auch direkt ausgeführt werden sollen, einen Block enthalten, der so oder so ähnlich aussieht:

```
if __name__ == "__main__":
    main()
```

Dieser Befehl weist Python an, `main()` auszuführen, wenn wir das Modul direkt aufrufen mit `python my_module.py`, aber den Code nicht auszuführen, wenn das Modul importiert wird. Diese Arten von Code-Blöcken werden häufig mit einer einfachen Pragma-Anweisung vom Testen ausgeschlossen:

```
if __name__ == "__main__":  # pragma: no cover
    main()
```

Damit wird Coverage.py angewiesen, entweder eine einzelne Zeile oder einen Code-Block auszuschließen. Wenn, wie in diesem Fall, das Pragma in der if-Anweisung steht, müsst ihr es nicht in beide Codezeilen einfügen.

Alternativ kann dies auch für alle Vorkommen konfiguriert werden:

```
[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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    ; 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
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
```

Siehe auch:

Configuration reference

16.6.2 Erweiterungen

In Coverage.py plugins findet ihr auch eine Reihe von Erweiterungen für Coverage.

16.6.3 Testabdeckung aller Tests mit GitHub-Actions

Nachdem ihr die Testabdeckung überprüft habt, könnt ihr die Dateien als GitHub-Action z.B. in einer `ci.yaml` als Artefakte hochladen um sie später in weiteren Jobs wiederverwenden zu können:

```yaml
      - name: Upload coverage data
        uses: actions/upload-artifact@v4
        with:
          name: coverage-data
          path: .coverage.*
          if-no-files-found: ignore
```

if-no-files-found: ignore
: ist sinnvoll, wenn nicht für alle Python-Versionen die Testabdeckung gemessen werden soll um schneller zum Ergebnis zu kommen. Daher solltet ihr nur für diejenigen Elemente eurer Matrix, die ihr berücksichtigen wollt, die Daten hochladen.

Nachdem alle Tests durchlaufen wurden, könnt ihr einen weiteren Job definieren, der die Ergebnisse zusammenführt:

```yaml
  coverage:
    name: Combine and check coverage
    needs: tests
    runs-on: ubuntu-latest
    steps:
      - name: Check out the repo
        uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: 3.12

      - name: Install dependencies
        run: |
          python -m pip install --upgrade coverage[toml]

      - name: Download coverage data
        uses: actions/download-artifact@v4
        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

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
82          # Report again and fail if under 100%.
83          python -Im coverage report --fail-under=100
84
85      - name: Upload HTML report if check failed
86        uses: actions/upload-artifact@v4
87        with:
88          name: html-report
89          path: htmlcov
90        if: ${{ failure() }}
```

needs: tests
: stellt sicher, dass alle Tests durchgeführt werden. Wenn euer Job, der die Tests ausführt, einen anderen Namen hat, müsst ihr ihn hier anpassen.

name: "Download coverage data"
: lädt die Daten der Testabdeckung herunter, die zuvor mit `name: "Upload coverage data"` hochgeladen wurden.

name: "Combine coverage and fail it it's under 100 %"
: kombiniert die Testabdeckung und erstellt einen HTML-Bericht, wenn die Bedingung `--fail-under=100` erfüllt ist.

Sobald der Workflow abgeschlossen ist, könnt ihr den HTML-Bericht herunterladen unter *YOUR_REPO* → *Actions* → *tests* → *Combine and check coverage*.

Siehe auch:

- How to Ditch Codecov for Python Projects
- structlog main.yml

16.6.4 Badge

Ihr könnt GitHub Actions verwenden, um ein Badge mit eurer Code-Coverage zu erstellen. Dabei wird zusätzlich ein GitHub Gist benötigt um die Parameter für das Badge, das von shields.io gerendert wird, zu speichern. Hierfür erweitern wir unsere `ci.yaml` folgendermaßen:

```
92      - name: Create badge
93        uses: schneegans/dynamic-badges-action@v1.7.0
94        with:
95          auth: ${{ secrets.GIST_TOKEN }}
96          gistID: YOUR_GIST_ID
97          filename: covbadge.json
98          label: Coverage
99          message: ${{ env.total }}%
100         minColorRange: 50
101         maxColorRange: 90
102         valColorRange: ${{ env.total }}
```

Zeile 97
: `GIST_TOKEN` ist ein persönliches GitHub-Zugangs-Token.

Zeile 98
: `YOUR_GIST_ID` solltet ihr durch eure eigene Gist-ID ersetzen. Falls ihr noch keine Gist-ID habt, könnt ihr diese erstellen mit:

1. Ruft https://gist.github.com auf und erstellt einen neuen Gist, den ihr z.B. `test.json` nennen könnt. Die ID des Gist ist der lange alphanumerische Teil der URL, den ihr hier benötigt.
2. Anschließend geht ihr zu https://github.com/settings/tokens und erstellt ein neues Token mit dem Gist-Bereich.
3. Geht schließlich zu *YOUR_REPO* → *Settings* → *Secrets* → *Actions* und fügt dieses Token hinzu. Ihr könnt ihm einen beliebigen Namen geben, z.B. *GIST_SECRET*.

   Wenn ihr Dependabot verwendet, um die Abhängigkeiten eures Repository automatisch zu aktualisieren, müsst ihr das *GIST_SECRET* auch in *YOUR_REPO* → *Settings* → *Secrets* → *Dependabot* hinzufügen.

Zeilen 102-104
Das Badge wird automatisch eingefärbt:

- 50 % in rot
- 90 % in grün
- mit einem Farbverlauf zwischen den beiden

Jetzt kann das Badge mit einer URL wie dieser angezeigt werden: `https://img.shields.io/endpoint?url=https://gist.githubusercontent.com/`*YOUR_GITHUB_NAME*`/`*GIST_SECRET*`/raw/covbadge.json`.

16.7 Mock

In diesem Kapitel werden wir die CLI testen. Hierfür werden wir das mock-Paket verwenden, das seit Python 3.3 als Teil der Python-Standardbibliothek unter dem Namen `unittest.mock` ausgeliefert wird. Für ältere Versionen von Python könnt ihr sie installieren mit:

```
$ . bin/activate
$ python -m pip install mock
```

```
C:> Scripts\activate.bat
C:> python -m pip install mock
```

Mock-Objekte werden manchmal auch als Test-Doubles, *Fakes* oder *Stubs* bezeichnet. Mit dem pytest-eigenen *monkeypatch*-Fixture und mock solltet ihr über alle Funktionen verfügen, die ihr benötigt.

16.7.1 Beispiel

Zunächst wollten wir mit einem einfachen Beispiel starten und überprüfen, ob die Arbeitstage von Montag bis Freitag korrekt ermittelt werden.

Zunächst importieren wir `datetime.datetime` und `Mock`:

```
1 from datetime import datetime
2 from unittest.mock import Mock
```

1. Dann definieren wir zwei Testtage:

```
monday = datetime(year=2021, month=10, day=11)
saturday = datetime(year=2021, month=10, day=16)
```

2. Nun definieren wir eine Methode zur Überprüfung der Arbeitstage, wobei die datetime-Bibliothek von Python Montage als `0` und Sonntage als `6` behandelt:

```
def is_workingday():
    today = datetime.today()
    return 0 <= today.weekday() < 5
```

3. Dann mocken wir datetime:

```
datetime = Mock()
```

4. Schließlich testen wir unsere beiden Mock-Objekte:

```
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 Testen mit Typer

Für die Tests der Items-CLI werden wir uns auch ansehen, wie der von Typer bereitgestellte `CliRunner` beim Testen hilft. Typer bietet eine Testschnittstelle, womit wir unsere Anwendung aufrufen können, ohne, wie in dem kurzen *capsys*-Beispiel auf `subprocess.run()` zurückgreifen zu müssen. Das ist gut, weil wir nicht simulieren können, was in einem separaten Prozess läuft. So können wir in `tests/cli/conftest.py` der `invoke()`-Funktion unseres `runner` nur unsere Anwendung `items.cli.app` und eine Liste von Strings übergeben, die den Befehl darstellt: genauer wandeln wir mit `shlex.split(command_string)()` die Befehle, z.B. `list -o "veit"` in `["list", "-o", "veit"]` um und können die Ausgabe dann abfangen und zurückgeben.

```
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)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        output = result.stdout.rstrip()
        return output

    return run_cli
```

Anschließend können wir diese Fixture einfach verwenden um z.B. die Version in `tests/cli/test_version.py` zu testen:

```
import items


def test_version(items_cli):
    assert items_cli("version") == items.__version__
```

16.7.3 Mocking von Attributen

Schauen wir uns an, wie wir Mocking verwenden können, um sicherzustellen, dass z.B. auch dreistellige Versionsnummern von `items.__version__()` korrekt über die CLI ausgegeben werden. Hierfür werden wir `mock.patch.object()` als Kontextmanager verwenden:

```
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 unserem Testcode importieren wir `items`. Das resultierende items-Objekt ist das, was wir patchen werden. Der Aufruf von `mock.patch.object()`, der als *Kontextmanager* innerhalb eines `with`-Blocks verwendet wird, gibt ein Mock-Objekt zurück, das nach dem `with`-Block aufgeräumt wird:

1. In diesem Fall wird das Attribut `__version__` von `items` für die Dauer des `with`-Blocks durch `"100.0.0"` ersetzt.
2. Anschließend verwenden wir `items_cli()`, um unsere CLI-Anwendung mit dem Befehl `"version"` aufzurufen. Wenn die Methode `version()` aufgerufen wird, ist das Attribut `__version__` jedoch nicht der ursprüngliche String, sondern der String, den wir mit `mock.patch.object()` ersetzt haben.

16.7.4 Mocking von Klassen und Methoden

In `src/items/cli.py` haben wir `config()` folgendermaßen definiert:

```
def config():
    """List the path to the Items db."""
    with items_db() as db:
        print(db.path())
```

`items_db()` ist ein *Kontextmanager*, der ein `items.ItemsDB`-Objekt zurückgibt. Das zurückgegebene Objekt wird dann als `db` verwendet, um `db.path()` aufzurufen. Wir sollten hier also zwei Dinge zu mocken: `items.ItemsDB` und eine seiner Methoden, `path()`. Beginnen wir mit der Klasse:

```
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)
```

Lasst und sicherstellen, dass es wirklich funktioniert:

```
$ 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 ===============================
```

Prima, nun müssen wir nur noch den Mock für die Datenbank in eine Fixture verschieben, denn wir werden ihn in vielen Testmethoden brauchen:

```
@pytest.fixture()
def mock_itemsdb():
    with mock.patch.object(items, "ItemsDB") as MockItemsDB:
        yield MockItemsDB.return_value
```

Diese Fixture mockt das `ItemsDB`-Objekt und gibt den `return_value` zurück, so dass Tests ihn verwenden können, um Dinge wie `path` zu ersetzen:

```
def test_mock_itemsdb(items_cli, mock_itemsdb):
    mock_itemsdb.path.return_value = "/foo/"
    result = runner.invoke(app, ["config"])
    assert result.stdout.rstrip() == "/foo/"
```

Alternativ kann zum Mocken von Klassen oder Objekten auch der `@mock.patch()`-Dekorator verwendet werden. In den folgenden Beispielen wird die Ausgabe von `os.listdir` gemockt. Dazu muss `db_path` nicht im Dateisystem vorhanden sein:

```
import os
from unittest import mock


@mock.patch("os.listdir", mock.MagicMock(return_value="db_path"))
def test_listdir():
    assert "db_path" == os.listdir()
```

Eine weitere Alternative ist, den Rückgabewert separat zu definieren:

```
@mock.patch("os.listdir")
def test_listdir(mock_listdir):
    mock_listdir.return_value = "db_path"
    assert "db_path" == os.listdir()
```

16.7.5 Mocks synchronisieren mit autospec

Mock-Objekte sind in der Regel als Objekte gedacht, die anstelle der echten Implementierung verwendet werden. Standardmäßig werden sie jedoch jeden Zugriff akzeptieren. Wenn das echte Objekt beispielsweise `start(index)()` zulässt, sollen unsere Mock-Objekte ebenfalls `start(index)()` zulassen. Dabei gibt es jedoch ein Problem. Mock-Objekte sind standardmäßig zu flexibel: sie würden auch `stort()` oder andere falsch geschriebene, umbenannte oder gelöschte Methoden oder Parameter akzeptieren. Dabei kann es im Laufe der Zeit zum SOG. Mock-Drift kommen, wenn sich die Schnittstelle, die ihr nachbildet, ändert, euer Mock in eurem Testcode jedoch nicht. Diese Form des Mock-Drifts kann durch das Hinzufügen von `autospec=True` zum Mock während der Erstellung gelöst werden:

```
@pytest.fixture()
def mock_itemsdb():
    with mock.patch.object(items, "ItemsDB", autospec=True) as MockItemsDB:
        yield MockItemsDB.return_value
```

Üblicherweise wird dieser Schutz mit `autospec` immer eingebaut. Die einzige mir bekannte Ausnahme ist, wenn die Klasse oder das Objekt, das gemockt wird, dynamische Methoden hat oder wenn Attribute zur Laufzeit hinzugefügt werden.

Siehe auch:

Die Python-Dokumentation hat einen großen Abschnitt über `autospec`: Autospeccing.

16.7.6 Aufruf überprüfen mit assert_called_with()

Bisher haben wir die Rückgabewerte einer Mocking-Methode verwendet, um sicherzustellen, dass unser Anwendungscode mit den Rückgabewerten richtig umgeht. Aber manchmal gibt es keinen nützlichen Rückgabewert, z.B. bei `items add some tasks -o veit`. In diesen Fällen können wir das Mock-Objekt fragen, ob es korrekt aufgerufen wurde. Nach dem Aufruf von `items_cli("add some tasks -o veit")()` wird nicht die API verwendet, um zu prüfen, ob das Element in die Datenbank gelangt ist, sondern ein Mock, um sicherzustellen, dass die CLI die richtige API-Methode korrekt aufgerufen hat. Die Implementierung des Befehls `add()` ruft schließlich `db.add_item()` mit einem `Item`-Objekt auf:

```
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)
```

Wenn `add_item()` nicht aufgerufen wird oder mit dem falschen Typ oder dem falschen Objektinhalt aufgerufen wird, schlägt der Test fehl. Wenn wir z.B. in `expected` den String `"Veit"` groß schreiben, aber nicht im CLI-Aufruf, erhalten wir folgende Ausgabe:

```
$ 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
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
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 ===============================
```

Siehe auch:

Es gibt eine ganze Reihe von Varianten von `assert_called()`. Eine vollständige Liste und Beschreibung erhaltet ihr in unittest.mock.Mock.assert_called.

Wenn die einzige Möglichkeit zum Testen darin besteht, den korrekten Aufruf sicherzustellen, erfüllen die verschiedenen `assert_called*()`-Methoden ihren Zweck.

16.7.7 Fehlerbedingungen erstellen

Lasst uns nun überprüfen, ob die Items-CLI Fehlerbedingungen korrekt behandelt. Hier ist z.B. die Implementierung des Löschbefehls:

```
@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}")
```

Um zu testen, wie die CLI mit einer Fehlerbedingung umgeht, können wir so tun, als ob `delete_item()` eine Exception erzeugt, indem wir dem Mock-Objekt die Exception dem Attribut side_effect des Mock-Objekts zuweisen, etwa so:

```
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
```

Das ist alles, was wir brauchen, um die CLI zu testen: Mocking von Rückgabewerten, Überprüfen der Aufrufe von Mock-Funktionen und das Mocking von Exceptions. Es gibt jedoch noch eine ganze Reihe weiterer Mocking-Techniken, die wir nicht behandelt haben. Lest also unbedingt unittest.mock — mock object library, wenn ihr Mocking ausgiebig nutzen möchtet.

16.7.8 Grenzen des Mocking

Eines der größten Probleme bei der Verwendung von Mocks besteht darin, dass wir bei in einem Test nicht mehr das Verhalten, sondern die Implementierung testen. Dies ist jedoch nicht nur zeitaufwändig, sondern auch gefährlich: Ein gültiges Refactoring z.B. das Ändern eines Variablennamens, kann Tests zum Scheitern bringen, wenn diese bestimmte Variable gemockt wurde. Wir wollen jedoch, dass unsere Tests nur dann fehlschlagen, wenn es Brüche im Verhalten gibt, nicht jedoch nur bei Codeänderungen.

Manchmal ist Mocking jedoch der einfachste Weg, Exceptions oder Fehlerbedingungen zu erzeugen und sicherzustellen, dass euer Code diese korrekt behandelt. Es gibt auch Fälle, in denen das Testen von Verhalten unzumutbar ist, wie z.B. beim Zugriff auf eine Zahlungs-API oder beim Senden von E-Mails. In diesen Fällen ist es eine gute Option zu testen, ob euer Code eine bestimmte API-Methode zum richtigen Zeitpunkt und mit den richtigen Parametern aufruft.

Siehe auch:

- Hynek Schlawack: "Don't Mock What You Don't Own"

16.7.9 Mocking vermeiden mit Tests auf mehreren Ebenen

Wir können die Items-CLI auch ohne Mocks testen indem wir auch die API verwenden. Dabei werden wir nicht die API testen, sondern sie nur verwenden, um das Verhalten von Aktionen zu überprüfen, die über die CLI ausgeführt werden. Das Beispiel *test_add_with_owner* können wir auch folgendermaßen testen:

```
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 testet die Implementierung der Befehlszeilenschnittstelle und stellt sicher, dass ein API-Aufruf mit bestimmten Parametern erfolgt. Beim Mixed-Layer-Ansatz wird das Verhalten getestet, um sicherzustellen, dass das Ergebnis unseren Vorstellungen entspricht. Diese Ansatz ist viel weniger ein Change-Detector und hat eine größere Chance, während eines Refactorings gültig zu bleiben. Interessanterweise sind die Tests auch etwa doppelt so schnell:

```
$ 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 ===============================
```

Wir könnten Mocking auch auf eine andere Weise vermeiden. Wir könnten das Verhalten vollständig über die CLI testen. Dazu müsste möglicherweise die Ausgabe der Items-Liste geparst werden, um den korrekten Datenbankinhalt zu überprüfen.

In der API gibt `add_item()` einen Index zurück und bietet eine `get_item(index)()`-Methode, die beim Testen hilft. Beide Methoden sind in der CLI nicht vorhanden, könnten es aber sein. Wir könnten vielleicht die Befehle `items get index` oder `items info index` hinzufügen, damit wir ein Item abrufen können, anstatt `items list für` alles verwenden zu müssen. `list` unterstützt auch bereits Filterung. Vielleicht würde das Filtern nach `index` funktionieren, anstatt einen neuen Befehl hinzuzufügen. Und wir könnten `items add` eine Ausgabe hinzufügen, die etwas sagt wie

Item hinzugefügt bei Index 3. Diese Änderungen würden in die Kategorie *Design for Testability* fallen. Sie scheinen auch keine tiefen Eingriffe in die Schnittstelle zu sein und sollten vielleicht in zukünftigen Versionen berücksichtigt werden.

16.7.10 Plugins zur Unterstützung von Mocking

Wir haben uns bisher auf die direkte Verwendung von mock konzentriert. Es gibt jedoch viele Plugins, die beim Mocking helfen, wie z.B. pytest-mock, das eine `mocker`-Fixture bereitstellt. Ein Vorteil ist, dass das Fixture nach sich selbst aufräumt, so dass ihr keinen `with`-Block verwenden müsst, wie wir es in unseren Beispielen getan haben.

Es gibt auch einige spezielle Mocking-Bibliotheken:

- Für das Mocking von Datenbankzugriffen eignen sich
  - pytest-postgresql
  - pytest-mongo
  - pytest-mysql
  - pytest-dynamodb.
- Zum Testen von HTTP-Servern könnt ihr pytest-httpserver verwenden.
- Zum Mocken von requests könnt ihr responses oder betamax verwenden.
- Weitere Tools für verschiedene Anforderungen sind
  - pytest-rabbitmq
  - pytest-solr
  - pytest-elasticsearch und pytest-redis.

16.8 tox

tox ist ein Automatisierungstool, das ähnlich wie ein *CI*-Tool funktioniert, aber sowohl lokal als auch in Verbindung mit anderen CI-Tools auf einem Server ausgeführt werden kann.

Im Folgenden richten wir uns tox für unsere Items-Anwendung so ein, dass es uns beim lokalen Testen hilft. Anschließend werden wir das Testen mithilfe von GitHub Actions einrichten.

16.8.1 Einführung in tox

tox ist ein Kommandozeilen-Tool, mit dem ihr eure komplette Testsuite in verschiedenen Umgebungen ausführen könnt. Wir werden tox verwenden, um das Items-Projekt in mehreren Python-Versionen zu testen. tox ist jedoch nicht nur auf Python-Versionen beschränkt. Ihr könnt es zum Testen mit verschiedenen Abhängigkeits-Konfigurationen und verschiedenen Konfigurationen für verschiedene Betriebssysteme verwenden. tox verwendet dabei Projektinformationen aus der `setup.py`- oder `pyproject.toml`-Datei für das zu testende Paket, um eine installierbare *Distribution eures Pakets* zu erstellen. Es sucht in der `tox.ini`-Datei nach einer Liste von Umgebungen, und führt dann jeweils folgende Schritte aus:

1. erstellt eine *virtuelle Umgebung*,
2. installiert einige Abhängigkeiten mit *pip*,
3. baut euer Paket,
4. installiert euer Paket mit pip,

5. führt weitere Tests aus.

Nachdem alle Umgebungen getestet wurden, gibt tox eine Zusammenfassung der Ergebnisse aus.

Bemerkung: Obwohl tox von vielen Projekten verwendet wird, gibt es Alternativen, die ähnliche Funktionen erfüllen. Zwei Alternativen zu tox sind nox und invoke.

16.8.2 tox einrichten

Bis jetzt hatten wir den items-Code in einem `src/`-Verzeichnis und die Tests in `tests/api/` und `tests/cli/`. Jetzt werden wir eine `tox.ini`-Datei hinzufügen, sodass die Struktur so aussieht:

```
items
├── ...
├── pyproject.toml
├── src
│   └── items
│       └── ...
├── tests
│   ├── api
│   │   ├── __init__.py
│   │   ├── conftest.py
│   │   └── test_....py
│   └── cli
│       ├── __init__.py
│       ├── conftest.py
│       └── test_....py
└── tox.ini
```

Dies ist ein typisches Layout für viele Projekte. Werfen wir einen Blick auf eine einfache `tox.ini`-Datei im Items-Projekt:

```
[tox]
envlist = py312

[testenv]
deps =
  pytest>=6.0
  faker
commands = pytest
```

Im `[tox]`-Abschnitt haben wir `envlist = py312` definiert. Dies ist eine Abkürzung, die tox anweist, unsere Tests mit Python Version 3.12 durchzuführen. Wir werden in Kürze weitere Python-Versionen hinzufügen, aber die Verwendung einer Version hilft, den Ablauf von tox zu verstehen.

Beachtet auch die Zeile `isolated_build = True`: Dies ist für alle mit `pyproject.toml` konfigurierten Pakete erforderlich. Für alle mit `setup.py`-konfigurierten Projekte, die die *setuptools*-Bibliothek verwenden, kann diese Zeile jedoch weggelassen werden.

Im `[testenv]`-Abschnitt werden unter `deps` `pytest` und `faker` als Abhängigkeiten aufgeführt. Somit weiß tox, dass wir diese beiden Werkzeuge zum Testen benötigen. Wenn ihr möchtet, könnt ihr auch angeben, welche Version verwendet werden soll, z.B. `pytest>=6.0`. Mit `commands` wird schließlich tox angewiesen, `pytest` in jeder Umgebung auszuführen.

16.8.3 tox ausführen

Bevor ihr tox ausführen könnt, müsst ihr sicherstellen, dass ihr es installiert:

```
$ python3 -m venv .
$ . bin/acitvate
$ python -m pip install tox
```

```
C:> python -m venv .
C:> Scripts\activate
C:> python -m pip install tox
```

Um tox auszuführen, startet einfach tox:

```
$ python -m tox
.pkg: _optional_hooks> python /Users/veit/cusy/prj/items_env/lib/python3.12/site-
↪packages/pyproject_api/_backend.py True hatchling.build
.pkg: get_requires_for_build_sdist> python /Users/veit/cusy/prj/items_env/lib/python3.12/
↪site-packages/pyproject_api/_backend.py True hatchling.build
.pkg: build_sdist> python /Users/veit/cusy/prj/items_env/lib/python3.12/site-packages/
↪pyproject_api/_backend.py True hatchling.build
py312: install_package> python -I -m pip install --force-reinstall --no-deps /Users/veit/
↪cusy/prj/items/.tox/.tmp/package/37/items-0.1.0.tar.gz
py312: commands[0]> pytest
============================= test session starts ==============================
...
configfile: pyproject.toml
testpaths: tests
plugins: Faker-25.0.0
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%]

============================== 49 passed in 0.09s ==============================
.pkg: _exit> python /Users/veit/cusy/prj/items_env/lib/python3.12/site-packages/
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪pyproject_api/_backend.py True hatchling.build
  py312: OK (1.40=setup[1.07]+cmd[0.33] seconds)
  congratulations :) (1.44 seconds)
```

16.8.4 Mehrere Python-Versionen testen

Hierfür erweitern wir `envlist` in der `tox.ini`-Datei um weitere Python-Versionen hinzuzufügen:

```
[tox]
envlist = py3{8,9,10,11,12}
isolated_build = True
skip_missing_interpreters = True
```

Damit werden wir jetzt Python-Versionen von 3.8 bis 3.12 testen. Zusätzlich haben wir auch die Einstellung `skip_missing_interpreters = True` hinzugefügt, damit tox nicht fehlschlägt, wenn auf eurem System eine der aufgeführten Python-Versionen fehlt. Ist der Wert auf `True` gesetzt, führt tox die Tests mit jeder verfügbaren Python-Version durch, überspringt aber Versionen, die es nicht findet, ohne fehlzuschlagen. Die Ausgabe ist sehr ähnlich, wobei ich in der folgenden Darstellung lediglich die Unterschiede hervorhebe:

```
 $ python -m tox
 py38: skipped because could not find python interpreter with spec(s): py38
 py38: SKIP  in 0.01 seconds
 py39: install_deps> python -I -m pip install faker 'pytest>=6.0'
 .pkg: _optional_hooks> python /Users/veit/cusy/prj/items_env/lib/python3.12/site-
↪packages/pyproject_api/_backend.py True hatchling.build
 .pkg: get_requires_for_build_sdist> python /Users/veit/cusy/prj/items_env/lib/python3.
↪12/site-packages/pyproject_api/_backend.py True hatchling.build
 .pkg: build_sdist> python /Users/veit/cusy/prj/items_env/lib/python3.12/site-packages/
↪pyproject_api/_backend.py True hatchling.build
 py39: install_package> python -I -m pip install --force-reinstall --no-deps /Users/veit/
↪cusy/prj/items/.tox/.tmp/package/34/items-0.1.0.tar.gz
 py39: commands[0]> pytest
 ============================= test session starts ==============================
 ...
 ============================= 49 passed in 0.15s ===============================
 py39: OK ✓ in 5.78 seconds
 py310: skipped because could not find python interpreter with spec(s): py310
 py310: SKIP  in 0.01 seconds
 py311: install_deps> python -I -m pip install faker 'pytest>=6.0'
 py311: install_package> python -I -m pip install --force-reinstall --no-deps /Users/
↪veit/cusy/prj/items/.tox/.tmp/package/35/items-0.1.0.tar.gz
 py311: commands[0]> pytest
 ============================= test session starts ==============================
 ...
 ============================= 49 passed in 0.11s ===============================
 py311: OK ✓ in 3.53 seconds
 py312: install_deps> python -I -m pip install faker 'pytest>=6.0'
 py312: install_package> python -I -m pip install --force-reinstall --no-deps /Users/
↪veit/cusy/prj/items/.tox/.tmp/package/36/items-0.1.0.tar.gz
 py312: commands[0]> pytest
 ============================= test session starts ==============================
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
...
============================== 49 passed in 0.11s ==============================
.pkg: _exit> python /Users/veit/cusy/prj/items_env/lib/python3.12/site-packages/
↪pyproject_api/_backend.py True hatchling.build
  py38: SKIP (0.01 seconds)
  py39: OK (5.78=setup[4.60]+cmd[1.18] seconds)
  py310: SKIP (0.01 seconds)
  py311: OK (3.53=setup[2.82]+cmd[0.71] seconds)
  py312: OK (3.23=setup[2.57]+cmd[0.67] seconds)
  congratulations :) (12.60 seconds)
```

16.8.5 Tox-Umgebungen parallel ausführen

Im vorherigen Beispiel wurden die verschiedenen Umgebungen nacheinander ausgeführt. Es ist auch möglich, sie mit der Option `-p` parallel laufen zu lassen:

```
$ python -m tox -p
py38: SKIP  in 0.07 seconds
py310: SKIP  in 0.07 seconds
py312: OK ✓ in 1.7 seconds
py311: OK ✓ in 1.75 seconds
  py38: SKIP (0.07 seconds)
  py39: OK (2.42=setup[2.01]+cmd[0.41] seconds)
  py310: SKIP (0.07 seconds)
  py311: OK (1.75=setup[1.35]+cmd[0.40] seconds)
  py312: OK (1.70=setup[1.30]+cmd[0.40] seconds)
  congratulations :) (2.47 seconds)
```

Bemerkung: Die Ausgabe ist nicht abgekürzt; dies ist die gesamte Ausgabe, die ihr seht, wenn alles funktioniert.

16.8.6 Coverage-Report in tox hinzufügen

Der `tox.ini`-Datei kann einfach die Konfiguration von Coverage Reports hinzugefügt werden. Dazu müssen wir `pytest-cov` zu den `deps`-Einstellungen hinzufügen, damit das `pytest-cov`-Plugin in den tox-Testumgebungen installiert wird. Das Einbinden von `pytest-cov` schließt auch alle seine Abhängigkeiten ein, wie z.B. Coverage. Wir erweitern dann `commands` zu `pytest --cov=items`:

```
[tox]
envlist = py3{8,9,10,11,12}
isolated_build = True
skip_missing_interpreters = True

[testenv]
deps =
 pytest>=6.0
 faker
commands = pytest
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
[testenv:coverage-report]
description = Report coverage over all test runs.
deps = coverage[toml]
skip_install = true
allowlist_externals = coverage
commands =
  coverage combine
  coverage report
```

Bei der Verwendung von Coverage mit `tox` kann es manchmal sinnvoll sein, in der `pyproject.toml`-Datei einen Abschnitt einzurichten, der Coverage mitteilt, welche Quelltextpfade als identisch betrachtet werden sollen:

```
[tool.coverage.paths]
source = ["src", ".tox/py*/**/site-packages"]
```

Der Items-Quellcode befindet sich zunächst in `src/items/`, bevor von tox die virtuellen Umgebungen erstellt und Items in der Umgebung installiert wird. Dann befindet es sich z.B. in `.tox/py312/lib/python3.12/site-packages/items`.

```
$ python -m tox
...
coverage-report: install_deps> python -I -m pip install 'coverage[toml]'
coverage-report: commands[0]> coverage combine
Combined data file .coverage.fay.local.74688.XgGaASxx
Skipping duplicate data .coverage.fay.local.74695.XmMjaOox
Skipping duplicate data .coverage.fay.local.74702.XUQUSgdx
coverage-report: commands[1]> coverage report
Name               Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------
src/items/api.py      68      1     16      1    98%   52
-------------------------------------------------------------
TOTAL                428      1    118      1    99%

27 files skipped due to complete coverage.
  py38: SKIP (0.02 seconds)
  py39: OK (6.24=setup[5.06]+cmd[1.18] seconds)
  py310: SKIP (0.01 seconds)
  py311: OK (3.97=setup[2.77]+cmd[1.20] seconds)
  py312: OK (3.80=setup[2.67]+cmd[1.13] seconds)
  coverage-report: OK (1.53=setup[0.90]+cmd[0.54,0.09] seconds)
  congratulations :) (15.59 seconds)
```

16.8.7 Mindestabdeckungsgrad festlegen

Bei der Ausführung der Coverage durch tox ist auch sinnvoll, einen Mindestabdeckungsgrad festzulegen, um eventuelle Ausrutscher bei der Coverage zu erkennen. Dies wird mit der Option `--cov-fail-under` erreicht:

```
$ python -m coverage report --fail-under=100
Name               Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------
src/items/api.py      68      1     16      1    98%   52
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
----------------------------------------------------------------
TOTAL                 428      1    118      1    99%

27 files skipped due to complete coverage.
Coverage failure: total of 99 is less than fail-under=100
```

Dadurch wird der Ausgabe die hervorgehobene Zeile hinzugefügt.

16.8.8 pytest-Parameter an tox übergeben

Wir können auch einzelne Tests mit tox aufrufen, indem wir eine weitere Änderung vornehmen, damit Parameter an pytest übergeben werden können:

```
[tox]
envlist =
    pre-commit
    docs
    py3{8,9,10,11,12}
    coverage-report
isolated_build = True
skip_missing_interpreters = True

[testenv]
extras =
  tests: tests
deps =
  tests: coverage[toml]
allowlist_externals = coverage
commands =
  coverage run -m pytest {posargs}
```

Um Argumente an pytest zu übergeben, fügt sie zwischen den tox-Argumenten und den pytest-Argumenten ein. In diesem Fall wählen wir `test_version`-Tests mit der Schlüsselwort-Option `-k` aus. Wir verwenden auch `--no-cov`, um die Abdeckung zu deaktivieren:

```
$ tox -e py312 -- -k test_version --no-cov
...
py312: commands[0]> coverage run -m pytest -k test_version --no-cov
============================= test session starts ==============================
...
configfile: pyproject.toml
testpaths: tests
plugins: cov-5.0.0, Faker-25.0.0
collected 49 items / 47 deselected / 2 selected

tests/api/test_version.py .                                              [ 50%]
tests/cli/test_version.py .                                              [100%]

======================= 2 passed, 47 deselected in 0.09s =======================
.pkg: _exit> python /Users/veit/cusy/prj/items_env/lib/python3.12/site-packages/
↪pyproject_api/_backend.py True hatchling.build
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
  py312: OK (2.22=setup[1.12]+cmd[1.10] seconds)
  congratulations :) (2.25 seconds)
```

`tox` eignet sich nicht nur hervorragend für die lokale Automatisierung von Testprozessen, sondern hilft auch bei Server-basierter *CI*. Fahren wir fort mit der Ausführung von `pytest` und `tox` mithilfe von GitHub-Aktionen.

16.8.9 `tox` mit GitHub-Aktionen ausführen

Wenn euer Projekt auf GitHub gehostet ist, könnt ihr GitHub-Actions verwenden um automatisiert eure Tests in verschiedenen Umgebungen ausführen zu können. Dabei sind eine ganze Reihe von Umgebungen für die GitHub-Actions verfügbar: github.com/actions/virtual-environments.

1. Um eine GitHub-Action in eurem Projekt zu erstellen, klickt auf *Actions* → *set up a workflow yourself*. Dies erstellt üblicherweise eine Datei `.github/workflows/main.yml`.
2. Gebt dieser Datei einen aussagekräftigeren Namen. Wir verwenden hierfür üblicherweise `ci.yml`.
3. Die vorausgefüllte YAML-Datei ist für unsere Zwecke wenig hilfreich. Ihr könnt hier einen `coverage`-Abschnitt einfügen, z.B. mit:

```yaml
jobs:
  coverage:
    name: Ensure 99% test coverage
    runs-on: ubuntu-latest
    needs: tests
    if: always()

    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          cache: pip
          python-version: 3.12

      - name: Download coverage data
        uses: actions/download-artifact@v4
        with:
          pattern: coverage-data-*
          merge-multiple: true

      - name: Combine coverage and fail if it's <99%.
        run: |
          python -m pip install --upgrade coverage[toml]
          python -m coverage combine
          python -m coverage html --skip-covered --skip-empty
          # Report and write to summary.
          python -m coverage report --format=markdown >> $GITHUB_STEP_SUMMARY
          # Report again and fail if under 99%.
          python -m coverage report --fail-under=99
```

name
: kann ein beliebiger Name sein. Er wird in der Benutzeroberfläche von GitHub Actions angezeigt.

`steps`
: ist eine Liste von Schritten. Der Name eines jeden Schrittes kann beliebig sein und ist optional.

`uses: actions/checkout@v4`
: ist ein GitHub-Actions-Tool, das unser Repository auscheckt, damit der Rest des Workflows darauf zugreifen kann.

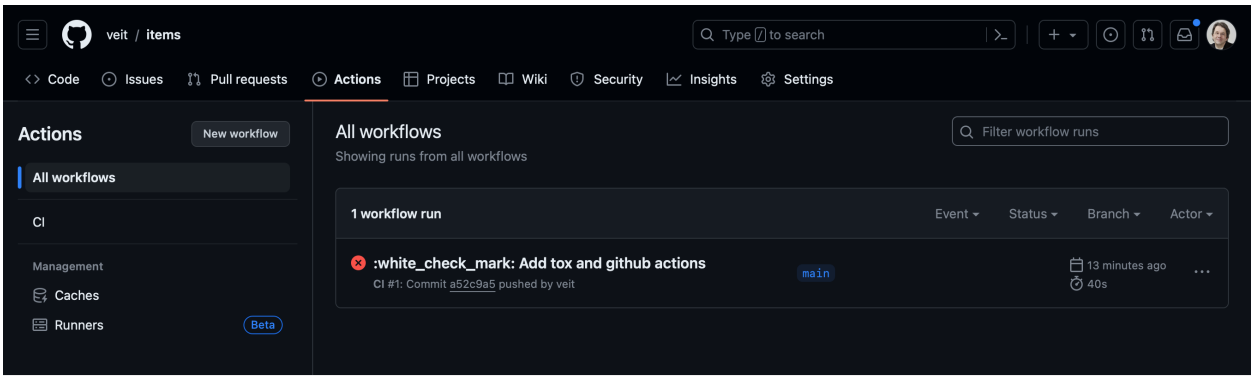
`uses: actions/setup-python@v5`
: ist ein GitHub-Actions-Tool, das Python konfiguriert und in einer Build-Umgebung installiert.

`with: python-version: ${{ matrix.python }}`
: sagt, dass eine Umgebung für jede der in `matrix.python` aufgeführten Python-Versionen erstellt werden soll.

4. Anschließend könnt ihr auf *Start commit* klicken. Da wir noch weitere Änderungen vornehmen wollen bevor die Tests automatisiert ausgeführt werden sollen, wählen wir *Create a new branch for this commit and start a pull request* und als Name für den neuen Branch `github-actions`. Schließlich könnt ihr auf *Create pull request* klicken.

5. Um nun in den neuen Branch zu wechseln, gehen wir zu *Code* → *main* → *github-actions*.

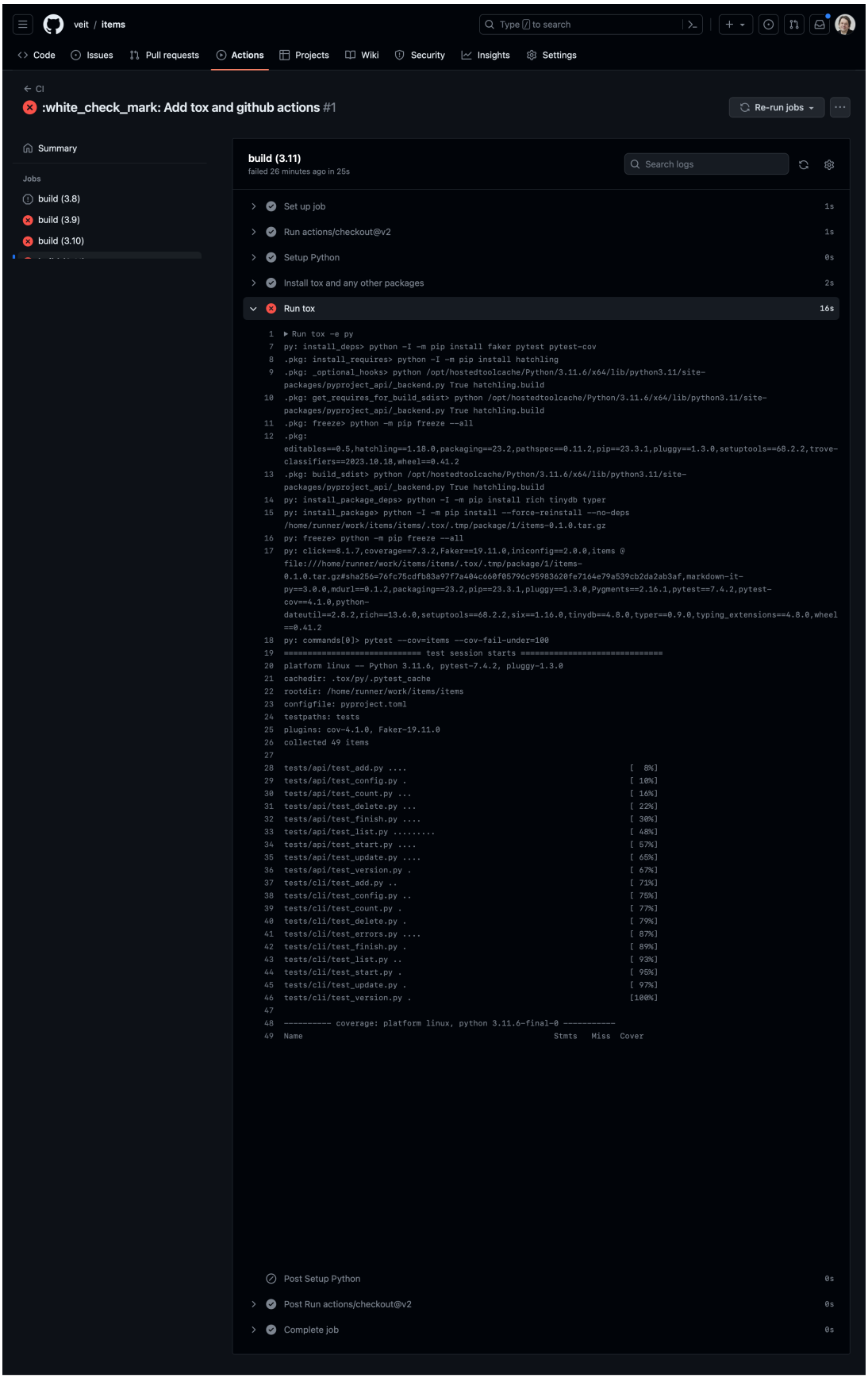
Die Actions-Syntax ist gut dokumentiert. Ein guter Startpunkt in der GitHub-Actions-Dokumentation ist die Seite Building and Testing Python. Die Dokumentation zeigt euch auch, wie ihr pytest direkt ohne tox ausführen könnt und wie ihr die Matrix auf mehrere Betriebssysteme erweitern könnt. Sobald ihr eure `*.yml`-Datei eingerichtet und in euer GitHub-Repository hochgeladen habt, wird sie automatisch ausgeführt. Im Reiter *Actions* könnt ihr anschließend die Durchläufe sehen:



Die verschiedenen Python-Umgebungen sind auf der linken Seite aufgelistet. Wenn ihr eine auswählt, werden die Ergebnisse für diese Umgebung angezeigt, wie im folgenden Screenshot dargestellt:

Siehe auch:

- Building and testing Python
- Workflow syntax for GitHub Actions


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
1 ▶ Run tox -e py
7 py: install_deps> python -I -m pip install faker pytest pytest-cov
8 .pkg: install_requires> python -I -m pip install hatchling
9 .pkg: _optional_hooks> python /opt/hostedtoolcache/Python/3.11.6/x64/lib/python3.11/site-packages/pyproject_api/_backend.py True hatchling.build
10 .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
11 .pkg: freeze> python -m pip freeze --all
12 .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
13 .pkg: build_sdist> python /opt/hostedtoolcache/Python/3.11.6/x64/lib/python3.11/site-packages/pyproject_api/_backend.py True hatchling.build
14 py: install_package_deps> python -I -m pip install rich tinydb typer
15 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
16 py: freeze> python -m pip freeze --all
17 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
18 py: commands[0]> pytest --cov=items --cov-fail-under=100
19 ============================= test session starts ==============================
20 platform linux -- Python 3.11.6, pytest-7.4.2, pluggy-1.3.0
21 cachedir: .tox/py/.pytest_cache
22 rootdir: /home/runner/work/items/items
23 configfile: pyproject.toml
24 testpaths: tests
25 plugins: cov-4.1.0, Faker-19.11.0
26 collected 49 items
27
28 tests/api/test_add.py .... [8%]
29 tests/api/test_config.py . [10%]
30 tests/api/test_count.py ... [16%]
31 tests/api/test_delete.py ... [22%]
32 tests/api/test_finish.py .... [30%]
33 tests/api/test_list.py .... [48%]
34 tests/api/test_start.py .... [57%]
35 tests/api/test_update.py .... [65%]
36 tests/api/test_version.py . [67%]
37 tests/cli/test_add.py .. [71%]
38 tests/cli/test_config.py .. [75%]
39 tests/cli/test_count.py . [77%]
40 tests/cli/test_delete.py . [79%]
41 tests/cli/test_errors.py .... [87%]
42 tests/cli/test_finish.py . [89%]
43 tests/cli/test_list.py .. [93%]
44 tests/cli/test_start.py . [95%]
45 tests/cli/test_update.py . [97%]
46 tests/cli/test_version.py . [100%]
47
48 ---------- coverage: platform linux, python 3.11.6-final-0 -----------
49 Name Stmts Miss Cover
Post Setup Python 0s
Post Run actions/checkout@v2 0s
Complete job 0s

16.8.10 Badge anzeigen

Nun könnt ihr in eurer `README.rst`-Datei noch ein Badge eures *CI*-Status hinzufügen, z.B. mit:

```
.. 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 Testabdeckung veröffentlichen

Die Testabdeckung könnt ihr auf GitHub veröffentlichen, s.a. *Coverage GitHub-Actions*.

16.8.12 tox erweitern

tox verwendet pluggy, um das Standardverhalten anzupassen. Pluggy findet ein Plugin, indem es nach einem Einstiegspunkt mit dem Namen `tox` sucht, z.B. in einer `pyproject.toml`-Datei:

```
[project.entry-points.tox]
my_plugin = "my_plugin.hooks"
```

Um das Plugin zu verwenden, muss es daher lediglich in der gleichen Umgebung installiert werden, in der auch tox läuft, und es wird über den definierten Einstiegspunkt gefunden.

Ein Plugin wird durch die Implementierung von Erweiterungspunkten in Form von Hooks erstellt. Der folgende Codeschnipsel würde zum Beispiel ein neues –my CLI definieren:

```
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")
```

Siehe auch:

- Extending tox
- tox development team

16.9 unittest2

unittest2 ist ein Backport von `unittest`, mit verbesserter API und besseren *Assertions* als in früheren Python-Versionen.

16.9.1 Beispiel

Möglicherweise wollt ihr das Modul unter dem Namen `unittest` importieren um die Portierung von Code auf neuere Versionen des Moduls in Zukunft zu vereinfachen:

```
import unittest2 as unittest


class MyTest(unittest.TestCase): ...
```

Auf diese Weise könnt ihr, wenn ihr zu einer neueren Python-Version wechselt und das Modul `unittest2` nicht mehr benötigt, einfach den Import in eurem Testmodul ändern, ohne dass ihr weiteren Code ändern müsst.

16.9.2 Installation

```
$ bin/python -m pip install unittest2
```

```
C:> Scripts\python -m pip install unittest2
```

16.10 Glossar

`assert`
: Ein Schlüsselwort, das die Codeausführung anhält, wenn sein Argument falsch ist.

Continuous Integration
CI
Kontinuierliche Integration
: Automatisches Überprüfen des Erstellungs- und Testprozesses auf verschiedenen Plattformen.

Dummy
: Objekt, das herumgereicht, aber nie wirklich benutzt. Normalerweise werden sie nur zum Füllen von Parameter-Listen verwendet.

`exception`
: Anpassbare Form von *assert*.

`except`
: Schlüsselwort, das verwendet wird, um eine *exception* abzufangen und sorgfältig zu behandeln.

Fake
: Objekt, das eine tatsächlich funktionierende Implementierung hat, in der Regel aber eine Abkürzung nehmen, die sie nicht für die Produktion geeignet macht.

Integrationstest
: Tests, die überprüfen, ob die verschiedenen Teile der Software wie erwartet zusammenarbeiten.

Mock
: Objekte, die mit *exception* programmiert sind, die eine Spezifikation der Aufrufe bilden, die ihr voraussichtlich erhalten werdet.

  Siehe auch:

  - Mock-Objekt

pytest
: Ein Python-Paket mit Test-Utilities.

Regressionstest
: Tests zum Schutz vor neuen Fehlern oder Regressionen, die durch neue Software und Updates auftreten können.

Stubs
: liefern vorgefertigte Antworten auf Aufrufe, die während des Tests getätigt werden, und reagieren in der Regel überhaupt nicht auf irgendetwas, das nicht für den Test programmiert wurde.

Test-driven development
TDD
Testgetriebene Entwicklung
: Eine Software-Entwicklungsstrategie, bei der die Tests vor dem Code geschrieben werden.

`try`
: Ein Schlüsselwort, das einen Teil des Codes schützt, der eine *exception* auslösen kann.

KAPITEL 17

Dokumentieren

Damit euer Software-Paket sinnvoll genutzt werden kann, sind Dokumentationen ierforderlich, die Beschreiben, wie eure Software installiert, betrieben, genutzt und verbessert werden kann:

- Diejenigen, die euer Paket nutzen wollen, benötigen Informationen,
  - welche Probleme eure Software löst und was die Hauptfunktionen und Limitationen der Software sind (`README`)
  - wie das Software beispielhaft verwendet werden kann
  - welche Veränderungen in aktuelleren Software-Versionen gekommen sind (`CHANGELOG`)
- Diejenigen, die die Software betreiben wollen, benötigen eine Installationsanleitung für eure Software und die erforderlichen Abhängigkeiten
- Diejenigen, die die Software verbessern wollen, benötigen Informationen
  - wie sie mit Fehlerbehebungen zur Verbesserung des Produkts beitragen können (`CONTRIBUTING`)
  - wie sie mit anderen kommunizieren (`CODE_OF_CONDUCT`) können

Alle gemeinsam benötigen Informationen, wie das Produkt lizenziert ist (`LICENSE`-Datei oder `LICENSES`-Ordner) und wie sie bei Bedarf Hilfe erhalten können.

Siehe auch:

- 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
- Cusy Design System: Schreiben

17.1 Erstellt ein Sphinx-Projekt

17.1.1 Installation und Start

1. Erstellt eine virtuelle Umgebung für euer Dokumentationsprojekt:

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

```
C:> python -m venv venv
```

2. Wechselt in die virtuelle Umgebung und installiert dort Sphinx:

```
$ 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. Erstellt euer Sphinx-Dokumentationsprojekt:

```
$ 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.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
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]:
> 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` ist die Ausgangsdatei für die Dokumentation, in der sich das Inhaltsverzeichnis befindet. Das Inhaltsverzeichnis kann von euch erweitert werden, sobald ihr neue `*.rst`-Dateien hinzufügt.

17.1.3 Generiert die Dokumentation

Ihr könnt die Dokumentation nun generieren, z.B. mit:

```
$ bin/sphinx-build -ab html docs/ docs/_build
```

```
C:> Scripts\sphinx-build -ab html docs\ docs\_build
```

a

generiert alle Seiten der Dokumentation neu.

Bemerkung: Dies ist immer dann sinnvoll, wenn ihr eurer Dokumentation neue Seiten hinzugefügt habt.

---

b

gibt an, welcher Builder zum Generieren der Dokumentation verwendet werden soll. In unserem Beispiel ist dies `html`.

17.2 reStructuredText

Siehe auch:

- reStructuredText Primer
- reStructuredText Quick Reference

17.2.1 Überschriften

```
Unterstreicht den Titel mit Interpunktionszeichen
=================================================

Ändert das Interpunktionszeichen für Untertitel
-----------------------------------------------
```

17.2.2 Absätze

Ein Absatz besteht aus einer oder mehreren Zeilen nicht eingerückten Textes.

Sie können durch Leerzeilen vom darüber und darunter liegenden Text getrennt werden.

```
Ein Absatz besteht aus einer oder mehreren Zeilen nicht eingerückten Textes.

Sie können durch Leerzeilen vom darüber und darunter liegenden Text getrennt
werden.
```

17.2.3 Inline-Auszeichnung

Kursiv, **fett** und `vorformattiert`

```
*Kursiv*, **fett** und ``vorformattiert``
```

17.2.4 Links

Externe Links

Hyperlink Link

```
`Hyperlink <http://en.wikipedia.org/wiki/Hyperlink>`_ `Link`_

.. _Link: http://en.wikipedia.org/wiki/Link_(The_Legend_of_Zelda)
```

Bemerkung: Vor einer mit `..` beginnenden Direktive muss immer eine Leerzeile stehen.

Interne Links

Stellen mit Querverweisen

Zu referenzierender Abschnitt

Um auf einen Abschnitt zu verweisen, verwendet *Zu referenzierender Abschnitt*.

```
.. _my-reference-label:

Zu referenzierender Abschnitt
.............................

Um auf einen Abschnitt zu verweisen, verwendet :ref:`my-reference-label`.
```

Auf Dokumente referenzieren

Link zur *Startseite* oder zu *Docstrings*.

```
Link zur :doc:`Startseite <../index>` oder zu :doc:`docstrings`.
```

Dokumente herunterladen

Link zu einem Dokument, das nicht von Sohinx gerendert werden soll, z.B. zu `autodoc-examples.rst`.

```
Link zu einem Dokument, das nicht von Sohinx gerendert werden soll,
:abbr:`z.B. (zum Beispiel)` zu :download:`docstrings-example.rst`.
```

17.2.5 Bilder

```
.. image:: uml/activity-diagram.svg
```

Andere semantische Auszeichnungen

Dateiliste

/Users/*NAME*/python-basics

```
:file:`/Users/{NAME}/python-basics`
```

Menüauswahlen und GUI-Beschriftungen

1. *Datei* → *Speichern unter* …
2. *Abschicken*

```
#. :menuselection:`Datei --> Speichern unter ...`
#. :guilabel:`&Abschicken`
```

17.2.6 Listen

17.2.7 Nummerierte Listen

1. Erstens
2. Zweitens
3. Drittens

```
#. Erstens
#. Zweitens
#. Drittens
```

Unnummerierte Listen

- Jeder Eintrag in einer Liste beginnt mit einem Asterisk (*)
- Listeneinträge können über mehrere Zeilen angezeigt werden, solange die Listeneinträge eingerückt bleiben.

```
* Jeder Eintrag in einer Liste beginnt mit einem Asterisk (``*``)
* Listeneinträge können über mehrere Zeilen angezeigt werden, solange die
  Listeneinträge eingerückt bleiben.
```

Definitionsliste

Term
: Definition des Begriffs

EIn anderer Term
: … und seine Definition

```
Term
    Definition des Begriffs
EIn anderer Term
   ... und seine Definition
```

17.2.8 Verschachtelte Listen

- Listen können auch verschachtelt werden
  - und Unterpunkte enthalten

```
* Listen können auch verschachtelt werden

  * und Unterpunkte enthalten
```

17.2.9 Literarische Blöcke

»Blockmarkierungen sehen aus wie Absätze, sind aber mit einem oder mehreren Leerzeichen eingerückt.«

```
»Blockmarkierungen sehen aus wie Absätze, sind aber mit einem oder mehreren
Leerzeichen eingerückt.«
```

17.2.10 Zeilenblöcke

Durch das Pipe-Zeichen wird dies zu einer Zeile.
Und dies wird eine weitere Zeile sein.

```
| Durch das Pipe-Zeichen wird dies zu einer Zeile.
| Und dies wird eine weitere Zeile sein.
```

17.2.11 Code-Blöcke

Codeblöcke werden mit zwei Doppelpunkten eingeleitet und eingerückt:

```
import docutils
print help(docutils)
```

```
>>> print 'Aber Doctests beginnen mit ">>>" und müssen nicht eingerückt werden.'
```

```
Codeblöcke werden mit zwei Doppelpunkten eingeleitet und eingerückt::

    import docutils
    print help(docutils)

>>> print 'Aber Doctests beginnen mit ">>>" und müssen nicht eingerückt werden.'
```

Siehe auch:

Code-Blöcke

17.2.12 Tabellen

Spaltenüberschrift	Spaltenüberschrift	Spaltenüberschrift	Spaltenüberschrift
Zeile 1, Spalte 1	Zeile 1, Spalte 2	Zeile 1, Spalte 3	Zeile 1, Spalte 4
Zeile 2, Spalte 1	Zeile 2, Spalte 2	Zeile 2, Spalte 3, colspan 2	
Zeile 3, Spalte 1	Zeile 3, Spalte 2	Zeile 3, Spalte 3, rowspan 2	Zeile 4, Spalte 4
Zeile 5, Spalte 1	Zeile 5, Spalte 2		Zeile 5, Spalte 4

```
+--------------------+--------------------+--------------------+--------------------+
| Spaltenüberschrift | Spaltenüberschrift | Spaltenüberschrift | Spaltenüberschrift |
+====================+====================+====================+====================+
| Zeile 1,           | Zeile 1,           | Zeile 1,           | Zeile 1,           |
| Spalte 1           | Spalte 2           | Spalte 3           | Spalte 4           |
+--------------------+--------------------+--------------------+--------------------+
| Zeile 2,           | Zeile 2,           | Zeile 2,                                |
| Spalte 1           | Spalte 2           | Spalte 3,  colspan 2                    |
+--------------------+--------------------+--------------------+--------------------+
| Zeile 3,           | Zeile 3,           | Zeile 3,           | Zeile 4,           |
| Spalte 1           | Spalte 2           | Spalte 3,          | Spalte 4           |
+--------------------+--------------------+ rowspan 2          +--------------------+
| Zeile 5,           | Zeile 5,           |                    | Zeile 5,           |
| Spalte 1           | Spalte 2           |                    | Spalte 4           |
+--------------------+--------------------+--------------------+--------------------+
```

17.2.13 Kommentare

```
.. Ein Kommentarblock beginnt mit zwei Punkten und kann weiter eingerückt werden.
```

17.3 Code-Blöcke

Code-Blöcke lassen sich mit der Direktive *code-block* sehr einfach darstellen. Zusammen mit Pygments hebt Sphinx dann die jeweilige Syntax automatisch hervor. Die passende Sprache für einen Code-Block könnt ihr angeben mit

`.. code-block::` LANGUAGE

Ihr könnt dies z.B. so verwenden:

```
.. code-block:: python

   import this
```

Optionen

:linenos:

Für *code-block* kann mit der `linenos`-Option auch angegeben werden, dass der Code-Block mit Zeilennummern angezeigt werden soll:

```
.. 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:

```
.. code-block:: python
   :lineno-start: 10

   import antigravity
```

```
10 import antigravity
```

:emphasize-lines:

`emphasize-lines` erlaubt euch, einzelne Zeilen hervorzuheben.

`.. literalinclude::` FILENAME

erlaubt euch, externe Dateien einzubinden.

Optionen

:emphasize-lines:

:linenos:

Hier ein Beispiel aus unserem 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
11     price: float
12     is_offer: Optional[bool] = None
13
14
15 @app.get("/")
16 def read_root():
17     return {"Hello": "World"}
18
19
20 @app.get("/items/{item_id}")
21 def read_item(item_id: int, q: Optional[str] = None):
22     return {"item_id": item_id, "q": q}
23
24
25 @app.put("/items/{item_id}")
26 def update_item(item_id: int, item: Item):
27     return {"item_name": item.name, "item_id": item_id}
```

:diff:

Wenn ihr das Diff eures Codes anzeigen möchtet, könnt ihr die alte Datei mit der `diff`-Option angeben, z.B.:

```
.. literalinclude:: main.py
   :diff: main.py.orig
```

```
--- /home/docs/checkouts/readthedocs.org/user_builds/python-basics-tutorial-de/
↪checkouts/latest/docs/document/main.py.orig
+++ /home/docs/checkouts/readthedocs.org/user_builds/python-basics-tutorial-de/
↪checkouts/latest/docs/document/main.py
@@ -1,12 +1,27 @@
 from typing import Optional
+
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
 from fastapi import FastAPI
+from pydantic import BaseModel

 app = FastAPI()
+
+
+class Item(BaseModel):
+    name: str
+    price: float
+    is_offer: Optional[bool] = None
+

 @app.get("/")
 def read_root():
     return {"Hello": "World"}


+
 @app.get("/items/{item_id}")
 def read_item(item_id: int, q: Optional[str] = None):
     return {"item_id": item_id, "q": q}
+
+
+@app.put("/items/{item_id}")
+def update_item(item_id: int, item: Item):
+    return {"item_name": item.name, "item_id": item_id}
```

17.3.1 Veralteter Code

`.. deprecated:: version`

Beschreibt, wann die Funktion veraltet wurde. Es kann auch eine Erklärung angegeben werden, um z.B. darüber zu informieren, was stattdessen verwendet werden sollte. Beispiel:

```
.. deprecated:: 4.1
   verwende stattdessen :func:`new_function`.
```

Veraltet ab Version 4.1: verwende stattdessen `new_function()`.

`:py:module:deprecated:`

Markiert ein Python-Modul als veraltet; es wird dann an verschiedenen Stellen als solches gekennzeichnet.

17.4 Platzhalter

Sphinx unterscheidet die folgenden Platzhaltervariablen:

`:envvar:`

Umgebungsvariable, die auch einen Verweis auf die passende `envvar`-Direktive erzeugt, falls sie existiert.

`:file:`

Der Name einer Datei oder eines Verzeichnisses. Geschweifte Klammern können verwendet werden, um z.B. einen variablen Teil anzugeben:

```
... is installed in :file:`/usr/lib/python3.{x}/site-packages` ...
```

In der generierten HTML-Dokumentation wird das `x` mit `em .pre` besonders ausgezeichnet und kursiv dargestellt, um zu zeigen, dass es durch die spezifische Python-Version ersetzt werden soll.

:makevar:

Der Name einer **make**-Variable

:samp:

Textbeispiel, wie z.B. Code, innerhalb dessen geschweifte Klammern verwendet werden können, um einen variablen Teil anzuzeigen, wie in *file* oder in `:samp:`print 1+{VARIABLE}``.

Ab Sphinx1.8 können geschweifte Klammern mit einem Backslash (\) angezeigt werden.

Bemerkung:

:content:

Diese Rolle hat standardmäßig keine besondere Bedeutung. Ihr könnt sie daher für alles Mögliche zu verwenden, z.B. auch für Variablennamen.

Siehe auch:

- Sphinx awesome sampdirective

17.5 UI-Elemente und Interaktionen

:guilabel:

Label, die als Teil einer interaktiven Benutzeroberfläche dargestellt werden, sollten mit *guilabel* gekennzeichnet werden. Jede in der Oberfläche verwendete Beschriftung sollte mit dieser Rolle gekennzeichnet werden, einschließlich Beschriftung von Schaltflächen, Fenstertiteln, Feldnamen, Menü- und Menüauswahlnamen und sogar Werte in Auswahllisten.

Ein Tastenkürzel für die GUI-Beschriftung kann mit einem et-Zeichen (&) eingefügt werden; dieses führt in der Ausgabe zur Unterstreichung des Folgebuchstabens.

Cancel erzielt ihr z.B. mit folgender Auszeichnung:

```
:guilabel:`&Cancel`
```

Bemerkung: Wenn ihr ein et-Zeichen einfügen wollt, könnt ihr es einfach verdoppeln.

:kbd:

Dies stellt eine Folge von Tasteneingaben dar. Welche Form die Tastenfolge hat, kann von plattform- oder anwendungsspezifischen Konventionen abhängen. Wenn es keine entsprechenden Konventionen gibt, sollten die Namen von Modifikatortasten ausgeschrieben werden, um die Zugänglichkeit zu verbessern. Auch sollte nicht auf eine bestimmte Tastaturbeschriftung referenziert werden.

`Ctrl-s` erzielt ihr z.B. mit folgender Auszeichnung:

```
:kbd:`Ctrl-s`
```

:menuselection:

Eine Menüauswahl sollte mit der Rolle `menuselection` markiert werden. Diese wird verwendet, um eine komplette Sequenz zu markieren, einschließlich der Auswahl von Untermenüs und der Auswahl bestimmter Operationein oder beliebiger Untersequenzen. Die Namen der einzelnen Auswahlen sollten durch `-->` getrennt werden.

View → *Cell Toolbar* → *Slideshow* erzielt ihr z.B. mit folgender Auszeichnung:

```
:menuselection:`View --> Cell Toolbar --> Slideshow`
```

menuselection unterstützt genau wie *guilabel* auch Tastaturkürzel mit einem et-Zeichen (&).

17.6 Weitere Direktiven

reStructuredText kann mit Directives erweitert werden. Sphinx macht hiervon ausgiebig Gebrauch. Hier sind einige Beispiele:

17.6.1 Inhaltsverzeichnis

Docstrings

Mit der Sphinx-Erweiterung sphinx.ext.autodoc können Docstrings auch in die Dokumentation aufgenommen werden. Die folgenden drei Direktiven können angegeben werden:

.. automodule::

.. autoclass::

.. autoefunction::

Diese dokumentieren ein Modul, eine Klasse oder eine Funktion unter Verwendung des jeweiligen Docstrings.

Installation

`sphinx.ext.autodoc` ist normalerweise bereits in der Sphinx-Konfigurationsdatei `docs/conf.py` angegeben:

```
extensions = ["sphinx.ext.autodoc", ...]
```

Wenn euer Paket und die zugehörige Dokumentation Teil desselben Repository sind, werden sie immer dieselbe relative Position im Dateisystem haben. In diesem Fall könnt ihr einfach die Sphinx-Konfiguration für `sys.path` bearbeiten, um den relativen Pfad zum Paket anzugeben, also:

```
sys.path.insert(0, os.path.abspath(".."))
import MODULE
```

Wenn ihr eure Sphinx-Dokumentation in einer virtuellen Umgebung installiert habt, könnt ihr euer Paket auch dort mitinstallieren, z.B. indem ihr es in eure `requirements.txt`-Datei eintragt.

Beispiele

Hier findet ihr einige Beispiele aus der Dokumentation des Python-`string`-Moduls:

```
``autodoc``-Beispiele
=====================

:py:mod:`string`-Modul
----------------------

.. automodule:: string

:py:class:`string.Template`-Klasse
----------------------------------

.. autoclass:: string.Template

:py:func:`string.capwords`-Funktion
-----------------------------------

.. autofunction:: string.capwords
```

Die Ausgabe ist autodoc-examples.

Bemerkung: Ihr solltet diese Richtlinien befolgen, wenn ihr Docstrings schreibt:

- **PEP 8#comments**
- **PEP 257#specification**

---

sphinx-autodoc-typehints

Mit **PEP 484** wurde eine Standardmethode für den Ausdruck von Typen in Python-Code eingeführt. Damit können Typen auch in Docstrings unterschiedlich ausgedrückt werden. Die Variante mit Typen nach PEP 484 hat den Vorteil, dass Typtester und IDEs zur statischen Codeanalyse eingesetzt werden können.

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
```

Typen 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
```

Bemerkung: 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

Die Sphinx-Erweiterung sphinx.ext.napoleon ermöglicht euch, verschiedene Abschnitte in Docstrings zu definieren, einschließlich:

- `Attributes`
- `Example`
- `Keyword Arguments`
- `Methods`
- `Parameters`
- `Warning`
- `Yield`

Es gibt zwei Arten von docstrings in `sphinx.ext.napoleon`:

- Google
- NumPy

Der Hauptunterschied besteht darin, dass Google Einrückungen verwendet und NumPy Unterstreichungen:

Google:

```
def func(arg1, arg2):
    """Summary line.

    Extended description of function.

    Args:
        arg1 (int): Description of arg1
        arg2 (str): Description of arg2
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    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
        Description of arg2

    Returns
    -------
    bool
        Description of return value

    """
    return True
```

Detaillierte Konfigurationsoptionen findet ihr in sphinxcontrib.napoleon.Config.

```
.. toctree::
   :maxdepth: 2

   start
   docstrings
```

Meta-Informationen

Autor des Abschnitts: Veit Schiele <veit@cusy.io>

Autor des Quellcode: Veit Schiele <veit@cusy.io>

```
.. sectionauthor:: Veit Schiele <veit@cusy.io>
.. codeauthor:: Veit Schiele <veit@cusy.io>
```

Bemerkung: Standardmäßig wird diese Angabe nicht in der Ausgabe berücksichtigt, bis ihr die Konfiguration für `show_authors` auf `True` setzt.

Siehe auch

Siehe auch:

Sphinx Directives

```
.. seealso::
    `Sphinx Directives
    <https://www.sphinx-doc.org/en/master/usage/restructuredtext/directives.html>`_
```

Glossar

Environment
: Eine Struktur, in der Informationen über alle Dokumente unterhalb des Stammverzeichnisses gespeichert und für Querverweise verwendet werden. Die Umgebung wird gespeichert, so dass nachfolgende Programmläufe nur neue und geänderte Dokumente lesen und parsen.

Quellverzeichnis
: Das Verzeichnis, das einschließlich seiner Unterverzeichnisse alle Quelldateien für ein Sphinx-Projekt enthält.

```
.. glossary::

   Environment
       Eine Struktur, in der Informationen über alle Dokumente unterhalb des
       Stammverzeichnisses gespeichert und für Querverweise verwendet werden.
       Die Umgebung wird gespeichert, so dass nachfolgende Programmläufe nur
       neue und geänderte Dokumente lesen und parsen.

   Quellverzeichnis
       Das Verzeichnis, das einschließlich seiner Unterverzeichnisse alle
       Quelldateien für ein Sphinx-Projekt enthält.
```

17.7 Intersphinx

sphinx.ext.intersphinx ermöglicht die Verknüpfung mit anderen Projektdokumentationen.

17.7.1 Konfiguration

In `docs/conf.py` muss Intersphinx als Erweiterung angegeben werden:

```
extensions = [..., "sphinx.ext.intersphinx"]
```

Externe Sphinx-Dokumentation kann dann angegeben werden, z.B. mit:

```
intersphinx_mapping = {
    "python": ("https://docs.python.org/3", None),
    "bokeh": ("https://bokeh.pydata.org/en/latest/", None),
}
```

Es können jedoch auch alternative Dateien für eine Bestandsaufnahme angegeben werden, z.B.:

```
intersphinx_mapping = {
    "python": ("https://docs.python.org/3", None, "python-inv.txt"),
}
```

17.7.2 Bestimmen von Linkzielen

Um die in einem Inventar verfügbaren Links zu ermitteln, könnt ihr z.B. Folgendes eingeben:

```
$ 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 Einen Link erstellen

Um einen Link auf https://docs.python.org/3/c-api/arg.html#c.PyArg_Parse zu erstellen, kann eine der folgenden Varianten angegeben werden:

PyArg_Parse()

```
:c:func:`PyArg_Parse`
```

PyArg_Parse()

```
:c:func:`!PyArg_Parse`
```

Parsing arguments

```
:c:func:`Parsing arguments <PyArg_Parse>`
```

17.7.4 Benutzerdefinierte Links

Ihr könnt auch eure eigenen `intersphinx`-Zuweisungen erstellen, wenn z.B. `objects.inv` Fehler aufweis wie bei Beautiful Soup.

Der Fehler kann mit korrigiert werden:

1. Installation of `sphobjinv`:

   ```
   $ python -m pip install sphobjinv
   ```

2. Dann könnt ihr die Originaldatei herunterladen mit:

   ```
   $ cd docs/build/
   $ mkdir _intersphinx
   $ !$
   $ curl -O https://www.crummy.com/software/BeautifulSoup/bs4/doc/objects.inv
   $ mv objects.inv bs4_objects.inv
   ```

3. Ändert die Sphinx-Konfiguration `docs/conf.py`:

```
intersphinx_mapping = {
    ...
    'bs4':    ('https://www.crummy.com/software/BeautifulSoup/bs4/doc/', "_
↪intersphinx/bs4_objects.inv")
}
```

4. Konvertiert die Datei in eine Textdatei:

```
$ sphobjinv convert plain bs4_objects.inv bs4_objects.txt
```

5. Editiert die Textdatei, z.B.:

```
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           -
```

   Diese Einträge können dann in einer Sphinx-Dokumentation mit referenziert werden:

```
- :class:`bs4.BeautifulSoup`
- :meth:`bs4.BeautifulSoup.get_text`
- :class:`bs4.element.Tag`
```

   Siehe auch:

   - Sphinx objects.inv v2 Syntax

6. Erstellt eine neue `objects.inv`-Datei:

```
$ sphobjinv convert zlib bs4_objects.txt bs4_objects.txt
```

7. Erstellt eine neue Sphinx-Dokumentation:

```
$ python -m sphinx -ab html docs/ docs/_build/
```

17.7.5 Rollen hinzufügen

Wenn ihr eine Fehlermeldung erhaltet, dass eine bestimmte Textrolle unbekannt ist, z.B.:

```
WARNING: Unknown interpreted text role "confval".
```

so könnt ihr sie in der `conf.py` hinzufügen:

```
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. Installiert plantuml:

```
$ sudo apt install plantuml
```

```
$ brew install plantuml
```

2. Installiert sphinxcontrib-plantuml:

```
$ bin/python -m pip install sphinxcontrib-plantuml
```

```
C:> Scripts\python -m pip install sphinxcontrib-plantuml
```

3. Konfiguriert Sphinx in der `conf.py`-Datei:

```
extensions = [..., "sphinxcontrib.plantuml"]

plantuml = "/PATH/TO/PLANTUML"
```

> **Bemerkung:** Auch in Windows werden in der Pfadangabe / angegeben.

Sequenzdiagramm

```
.. uml::

   Browser -> Server: Authentifizierungsanfrage
   Server --> Browser: Authentifizierungsantwort

   Browser -> Server: Eine andere Authentifizierungsanfrage
   Browser <-- Server: Eine andere Authentifizierungsantwort
```

`->`
: wird verwendet, um eine Nachricht zwischen zwei Akteuren zu zeichnen. Die Akteure müssen nicht explizit deklariert werden.

`-->`
: wird verwendet, um eine gepunktete Linie zu zeichnen.

`<- und <--`
: verändert die Zeichnung nicht, kann aber die Lesbarkeit erhöhen.

> **Bemerkung:** Dies gilt nur für Sequenzdiagramme. In anderen Diagrammen können andere Regeln gelten.

Anwendungsfall-Diagramm

```
.. uml::

   :Nutzende Person: --> (Verwendung)
   "Gruppe von\nAdministratoren" as Admin
   "Verwenden der\nAnwendung" as (Verwendung)
   Admin --> (Administrieren\nder Anwendung)
```

Anwendungsfälle werden von runden Klammern `()` umschlossen und ähneln einem Oval.

Alternativ kann auch das Schlüsselwort `usecase` verwendet werden, um einen Anwendungsfall zu definieren. Darüber hinaus ist es möglich, mit dem Schlüsselwort `as` einen Alias zu definieren. Dieser Alias kann dann bei der Definition von Beziehungen verwendet werden.

Mit `\n` könnt ihr Zeilenumbrüche in den Namen der Anwendungsfälle einfügen.

Aktivitätsdiagramm

`(*)`
: Start- und Endknoten eines Aktivitätsdiagramms.

  `(*top)`
  : In einigen Fällen kann dies verwendet werden um den Startpunkt an den Anfang eines Diagramms zu verschieben.

`-->`
: definiert eine Aktivität

  `-down->`
  : Pfeil nach unten (Standardwert)

  `-right-> or ->`
  : Pfeil nach rechts

  `-left->`
  : Pfeil nach links

  `-up->`
  : Pfeil nach oben

`if`, `then`, `else`
: Schlüsselworte für die Definition von Verzweigungen.

  Beispiel:

  ```
  .. uml::

     (*) --> "Initialisierung"
     if "ein Test" then
     -->[wahr] "Eine Aktivität"
     --> "Eine andere Aktivität"
     -right-> (*)
     else
     ->[falsch] "Etwas anderes"
     -->[Ende des Prozesses] (*)
     endif
  ```

fork, fork again und end fork oder end merge
Schlüsselworte für die parallele Verarbeitung.

Beispiel:

```
.. uml::

   start
   fork
     :Aktion 1;
   fork again
     :Aktion 2;
   end fork
   stop
```

Klassendiagramm

`abstract class`, `abstract`

Beispiel:

```
.. uml::

    abstract class  "Abstrakte Klasse"
```

`annotation`

Beispiel:

```
.. uml::

    annotation      Anmerkung
```

`circle`, `()`

Beispiel:

```
.. uml::

    circle          Kreis
```

`class`

Beispiel:

```
.. uml::

    class           Klasse
```

diamond, <>
An empty diamond stands for an association, a black diamond for a composition.

Beispiel:

```
.. uml::

   diamond         Assoziation
```

`entity`

Beispiel:

```
.. uml::

   entity          Entität
```

`enum`

Beispiel:

```
.. uml::

   enum            Aufzählung
```

`interface`

Beispiel:

```
.. uml::

   interface       Schnittstelle
```

17.9 Erweiterungen

Siehe auch:

Sphinx Extensions

17.9.1 Eingebaute Erweiterungen

sphinx.ext.autodoc
bindet Dokumentationen aus Docstrings ein.

sphinx.ext.autosummary
erzeugt Zusammenfassungen von Funktionen, Methoden und Attributen aus Docstrings.

sphinx.ext.autosectionlabel
referenziert Abschnitte mit Hilfe des Titels.

sphinx.ext.graphviz
rendert Graphviz Graphen.

sphinx.ext.ifconfig
schließt Inhalte nur unter bestimmten Bedingungen ein.

sphinx.ext.intersphinx
ermöglicht das Einbinden anderer Projektdokumentationen.

sphinx.ext.mathjax
Rendertmathematische Formeln über JavaScript.

sphinx.ext.napoleon
unterstützt NumPy und Google Style Docstrings.

sphinx.ext.todo
unterstützt ToDo-Elemente.

sphinx.ext.viewcode
fügt Links auf den Quellcode der Sphinx-Dokumentation hinzu.

Siehe auch:

sphinx/sphinx/ext/

17.9.2 Erweiterungen von Drittanbietern

nbsphinx
Jupyter Notebooks in Sphinx

jupyter-sphinx
ermöglicht das Rendern von interaktiven Jupyter-Widgets in Sphinx.

Siehe auch:

Embedding Widgets in the Sphinx HTML Documentation.

Breathe
ReStructuredText and Sphinx bridge to Doxygen

numpydoc
NumPy Sphinx-Erweiterung.

Releases
schreibt eine Changelog-Datei.

sphinxcontrib-napoleon
Präprozessor zum Parsen von NumPy- und Google-Style Docstrings.

sphinx-autodoc-annotation
verwendet Python3-Annotations in Sphinx docstrings

Sphinx-autodoc-typehints
Unterstützung von Typ-Hints für die Sphinx-Autodoc-Erweiterung.

sphinx-git
git-Changelog für Sphinx.

Sphinx Gitstamp Generator Extension
fügt git Zeitstempel im Kontext ein

sphinx-intl
Sphinx-Erweiterung für Übersetzungen.

sphinx-autobuild
überwacht ein Sphinx-Repository und erstellt neue Dokumentation, sobald Änderungen vorgenommen werden.

Sphinx-Needs
erlaubt die Definition, Verlinkung und Filterung von need-Objekten, also z.B. Anforderungen und Testfälle

Sphinx-pyreverse
erstellt ein UML-Diagramm von Python-Modulen

sphinx-jsonschema
zeigt ein JSON Schema in der Sphinx-Dokumentation

Sphinxcontrib-mermaid
ermöglicht euch, Mermaid-Grafiken in Ihre Dokumente einzubetten.

Sphinx Sitemap Generator Extension
generiert multiversion- und multilanguage sitemaps für die HTML-Version

Sphinx Lint
basiert auf rstlint.py aus CPython.

sphinx-toolbox
Werkzeugkasten für Sphinx mit vielen nützlichen Werkzeugen.

Siehe auch:

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 Eigene Erweiterungen

Lokale Erweiterungen in einem Projekt sollten relativ zur Dokumentation angegeben werden. Der entsprechende Pfad wird in der Sphinx-Konfigurationsdatei `docs/conf.py` angegeben. Wenn sich eure Erweiterung im Verzeichnis `exts` in der Datei `foo.py` befindet, dann sollte die `conf.py`-Datei wie folgt aussehen:

```
import sys
import os

sys.path.insert(0, os.path.abspath("exts"))

extensions = ["foo", ...]
```

Siehe auch:

- Developing extensions for Sphinx
- Application API

17.10 Testen

17.10.1 Formatierung

Ob die *Sphinx*-Dokumentation in gültigem *reStructuredText*-Format geschrieben ist, lässt sich mit sphinx-lint überprüfen. Dies binden wir üblicherweise in unsere pre-commit-Konfiguration ein:

Quellcode 1: .pre-commit-config.yaml

```
- repo: https://github.com/sphinx-contrib/sphinx-lint
  rev: v0.9.1
  hooks:
    - id: sphinx-lint
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
      args: [--jobs=1]
      types: [rst]
```

Siehe auch:

Mit Sybil könnt ihr nicht nur *reStructuredText* überprüfen, sondern z.B. auch Markdown und Myst. Darüberhinaus kann Sybil auch Code-Blöcke in der Dokumentation entweder mit *pytest* oder mit *Unittest* überprüfen.

17.10.2 Code-Formatierung

Die Formatierung von Code-Blöcken lässt sich mit blacken-docs überprüfen, das Black verwendet. Üblicherweise binden wir die Bibliothek über das pre-commit-Framework ein:

```
- repo: https://github.com/adamchainz/blacken-docs
  rev: "v1.12.1"
  hooks:
  - id: blacken-docs
    additional_dependencies:
    - black
```

blacken-docs unterstützt aktuell die folgenden black-Optionen:

- line-length
- preview
- skip-string-normalization
- target-version

17.10.3 Build-Fehler

Ihr habt die Möglichkeit, vor der Veröffentlichung eurer Änderungen zu überprüfen, ob eure Inhalte ordnungsgemäß erstellt werden. Hierfür hat Sphinx einen pingelig (ENGL. nitpicky)-Modus, der mit der Option `-n` aufgerufen werden kann, also z.B. mit:

```
$ bin/python -m sphinx -nb html docs/ docs/_build/
```

```
C:> Scripts\python -m sphinx -nb html docs\ docs\_build\
```

17.10.4 Links überprüfen

Ihr könnt auch automatisiert sicherstellen, dass die von euch angegebenen Linkziele erreichbar sind. Unser Dokumentationswerkzeug Sphinx verwendet hierfür einen `linkcheck`-Builder, den ihr GGF. (gegebenenfalls) aufrufen könnt mit:

```
$ bin/python -m sphinx -b linkcheck docs/ docs/_build/
```

```
C:> Scripts\python -m sphinx -b linkcheck docs\ docs\_build\
```

Die Ausgabe kann dann z.B. so aussehen:

```
$ 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/
(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
```

17.10.5 Kontinuierliche Integration

GGF. (Gegebenenfalls) könnt ihr auch automatisiert in eurer *CI*-Pipeline überprüfen, ob die Dokumentation gebaut wird und die Links gültig sind. In *tox* kann die Konfiguration folgendermaßen ergänzt werden:

Quellcode 2: tox.ini

```
[testenv:docs]
# Keep base_python in sync with ci.yml and .readthedocs.yaml.
base_python = py312
extras = docs
commands =
  sphinx-build -n -T -W -b html -d {envtmpdir}/doctrees docs docs/_build/html
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
[testenv:docs-linkcheck]
base_python = {[testenv:docs]base_python}
extras = {[testenv:docs]extras}
commands = sphinx-build -W -b linkcheck -d {envtmpdir}/doctrees docs docs/_build/html
```

Anschließend könnt ihr z.B. für GitHub folgende Jobs definieren:

Quellcode 3: .github/workflows/ci.yml

```
docs:
  name: Build docs and run doctests
  needs: build-package
  runs-on: ubuntu-latest
  steps:
  - name: Download pre-built packages
    uses: actions/download-artifact@v4
    with:
      name: Packages
      path: dist
  - run: tar xf dist/*.tar.gz --strip-components=1

  - uses: actions/setup-python@v5
    with:
      # Keep in sync with tox.ini/docs and .readthedocs.yaml
      python-version: "3.12"
      cache: pip
  - run: python -m pip install tox
  - run: python -m tox run -e docs
```

17.11 shot-scraper

shot-scraper ist ein Werkzeug, mit dem sich der Prozess der Aktualisierung von Screenshots automatisieren lässt.

17.11.1 Installation

```
$ python -m pip install shot-scraper
$ shot-scraper install
```

Bemerkung: Die zweite Zeile installiert den benötigten Browser.

17.11.2 Verwendung

shot-scraper kann auf zweierleis Art verwendet werden

1. …für einzelne Screenshots auf der Kommandozeile:

```
$  shot-scraper https://jupyter-tutorial.readthedocs.io/de/latest/clean-prep/index.
↪html -o ~/Downloads/clean-prep.png
```

   …oder mit zusätzlichen Optionen, z.B. für JavaScript- und CSS-Selektoren:

```
$ shot-scraper https://jupyter-tutorial.readthedocs.io/de/latest/clean-prep/
↪index.html -s '#overview' -o ~/Downloads/clean-prep.png
```

2. …für eine Reihe von Screenshots, die in einer YAML-Datei konfiguriert sind:

```
- 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
```

   Anschließend kann `shot-scraper multi` verwendet werden, z.B.:

```
$ 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'
```

   Siehe auch:

   - In der README.md-Datei findet ihr eine vollständige Übersicht über die möglichen Optionen.
   - Im shot-scraper-demo-Repository findet ihr eine deutlich umfangreichere shots.yaml-Datei.

17.11.3 GitHub-Actions

shot-scraper lässt sich einfach in GitHub Actions einbinden. Im shot-scraper-demo-Repository findet sich auch eine examplarische shots.yml. Einmal am Tag werden zwei Screenshots erzeugt und zurück in das Repository übertragen. Beachtet jedoch, dass das Speichern von Bilddateien, die sich häufig ändern, die Revisionshistorie sehr unleserlich machen können. Daher solltet ihr shot-scraper mit Bedacht zusammen mit GitHub Actions verwenden.

17.12 Badges

Einige dieser Informationen und mehr können als Badges abgerufen werden. Sie sind hilfreich, um einen schnellen Überblick über ein Produkt zu erhalten. Für das cookiecutter-namespace-template sind dies z.B.:

Ihr könnt auch eigene Badges erstellen, z.B.:

Siehe auch:

- shields.io

17.13 Sphinx

Für umfangreiche Dokumentationen könnt ihr z.B. Sphinx verwenden, ein Dokumentationswerkzeug, das reStructuredText in HTML oder PDF, EPub und man pages umwandelt. Auch die Python Basics werden mit Sphinx erstellt. Um einen ersten Eindruck von Sphinx zu bekommen, könnt ihr euch den Quellcode dieser Seite unter dem Link Page source ansehen.

Ursprünglich wurde Sphinx für die Dokumentation von Python entwickelt und wird heute in fast allen Python-Projekten verwendet, darunter NumPy and SciPy, Matplotlib, Pandas und SQLAlchemy.

Die Sphinx autodoc-Funktion, die zur Erstellung von Dokumentation aus Python-*Docstrings* verwendet werden kann, könnte ebenfalls zur Verbreitung von Sphinx unter Python-Entwicklern beitragen. Insgesamt ermöglicht es Sphinx Entwicklungsteams, eine vollständige Dokumentation an Ort und Stelle zu erstellen. Oft wird die Dokumentation auch im gleichen Git-Repository gespeichert, so dass die Erstellung der neuesten Software-Dokumentation einfach bleibt.

Sphinx wird auch in Projekten außerhalb der Python-Gemeinschaft eingesetzt, z.B. für die Dokumentation des Linux-Kernels: Kernel documentation update.

Read the Docs wurde entwickelt, um die Dokumentation weiter zu vereinfachen. Read the Docs erleichtert das Erstellen und Veröffentlichen von Dokumentationen nach jedem Commit.

Für die Projektdokumentation kann die Visualisierung von Git Feature Branches und Tags mit git-big-picture hilfreich sein.

Bemerkung: Wenn der Inhalt von `long_description` in `setup()` in reStructured Text geschrieben ist, wird er als gut formatiertes HTML im *Python Package Index* (*PyPI*) angezeigt.

17.14 Andere Dokumentationswerkzeuge

Pycco
: ist eine Python-Portierung von Docco.

KAPITEL 18

Anhang

18.1 Reguläre Ausdrücke

Siehe auch:

- www.regular-expressions.info
- AutoRegex

18.1.1 []

Eckige Klammern definieren eine Liste oder einen Bereich von zu suchenden Zeichen:

[abc]
: entspricht a, b oder c

[a-z]
: entspricht jedem Kleinbuchstaben

[A-Za-z]
: entspricht jedem Buchstaben

[A-Za-z0-9]
: entspricht einem beliebigen Buchstaben oder einer beliebigen Ziffer

18.1.2 Anzahl

.
: entspricht einem einzelnen Zeichen

*****
: entspricht null oder mehr Mal dem vorhergehenden Element, z.B. `colou*r` passt zu `color`, `colour`, `colouur`, usw.

?
: entspricht null oder einmal dem vorhergehenden Element. `colou?r` passt zu `color` und `colour`

+
: entspricht ein- oder mehr Mal dem vorhergehenden Element, z.B. `.+` passt zu `.`, `..`, `...` usw.

{N}
: entspricht `N` Mal dem vorhergehenden Element.

{N,}
: entspricht `N` oder mehr Mal dem vorhergehenden Element.

{N,M}
: entspricht mindestens `N` mal dem vorhergehenden Element, aber nicht mehr als `M` mal.

18.1.3 Position

^
: setzt die Position an den Anfang der Zeile.

$
: setzt die Position an das Ende der Zeile.

18.1.4 Verknüpfung

|
: logisches *oder*.

18.1.5 Escape-Zeichen und Literale

****
: wird verwendet, um nach einem Sonderzeichen zu suchen, z.B. um `.org` zu finden, müsst ihr den regulären Ausdruck `\.org` verwenden, da `.` das Sonderzeichen ist, das auf jedes Zeichen passt.

\d
: passt zu jeder einzelnen Ziffer.

\w
: passt auf jeden Teil eines Wortzeichens und ist äquivalent zu `[A-Za-z0-9]`.

\s
: passt zu jedem Leerzeichen, Tabulator oder Zeilenumbruch.

\b
: passt zu einem Muster an einer Wortgrenze.

18.2 Unicode und Zeichenkodierungen

Es gibt Dutzende von Zeichenkodierungen. Einen Überblick über die Encodings von Python erhaltet ihr in Encodings and Unicode.

18.2.1 Das `string`-Modul

Das string-Modul von Python unterscheidet die folgenden String-Konstanten, die alle in den ASCII-Zeichensatz fallen:

```
# 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"
punctuation = r"""!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"""
printable = digits + ascii_letters + punctuation + whitespace
```

Die meisten dieser Konstanten sollten in ihrem Bezeichnernamen selbsterklärend sein. `hexdigits` und `octdigits` beziehen sich auf die Hexadezimal- bzw. (beziehungsweise) Oktalwerte. Ihr könnt diese Konstanten für alltägliche String-Manipulation verwenden:

```
>>> import string
>>> hepy = "Hello Pythonistas!"
>>> hepy.rstrip(string.punctuation)
'Hello Pythonistas'
```

Das string-Modul arbeitet jedoch standardmäßig mit Unicode, der als Binärdaten (Bytes) dargestellt wird.

18.2.2 Unicode

Es ist offensichtlich, dass der ASCII-Zeichensatz nicht annähernd groß genug ist, um alle Sprachen, Dialekte, Symbole und Glyphen zu erfassen; er ist nicht einmal groß genug für das Englische.

ASCII ist zwar eine vollständige Untermenge von Unicode – die ersten 128 Zeichen in der Unicode-Tabelle entsprechen genau den ASCII-Zeichen – Unicode umfasst jedoch eine viel größere Menge von Zeichen. Dabei ist Unicode selbst keine Kodierung sondern wird durch verschiedene Zeichenkodierungen implementiert wobei UTF-8 das vermutlich am häufigsten verwendete Kodierungsschema ist.

Bemerkung: Die Python-Hilfedokumentation hat einen Eintrag für Unicode: gebt `help()` und dann `UNICODE` ein. Es wird ausführlich auf die verschiedenen Möglichkeiten, Python-Strings zu erstellen, eingegangen.

Siehe auch:

- Unicode HOWTO
- What's New In Python 3.0: Text Vs. Data Instead Of Unicode Vs. 8-bit

Unicode und UTF-8

Während Unicode ein abstrakter Kodierungsstandard ist, ist UTF-8 ein konkretes Kodierungsschema. Der Unicode-Standard ist eine Zuordnung von Zeichen zu Codepunkten und definiert mehrere verschiedene Kodierungen aus einem einzigen Zeichensatz. UTF-8 ist ein Kodierungsschema für die Darstellung von Unicode-Zeichen als Binärdaten mit einem oder mehreren Bytes pro Zeichen.

18.2.3 Kodierung und Dekodierung in Python 3

Der str-Typ ist für die Darstellung von menschenlesbarem Text gedacht und kann alle Unicode-Zeichen enthalten. Der bytes-Typ hingegen repräsentiert Binärdaten, die nicht von vornherein mit einer Kodierung versehen sind. `str.encode()` und `bytes.decode()` sind die Methoden des Übergangs vom einen zum anderen:

```
>>> "schön".encode("utf-8")
b'sch\xc3\xb6n'
>>> b"sch\xc3\xb6n".decode("utf-8")
'schön'
```

Das Ergebnis von `str.encode()` ist ein Bytes-Objekt. Sowohl Bytes-Literale (wie `b'sch\xc3\xb6n'`) als auch die Darstellungen von Bytes lassen nur ASCII-Zeichen zu. Aus diesem Grund darf beim Aufruf von `"schön".encode("utf-8")` das ASCII-kompatible `"sch"` so dargestellt werden, wie es ist, das ö wird jedoch zu `"\xc3\xb6"`. Diese chaotisch aussehende Sequenz repräsentiert zwei Bytes, `c3` und `b6` als Hexadezimalwerte.

Tipp: In `.encode()` und `.decode()` ist der Kodierungsparameter standardmäßig `"utf-8"`; dennoch empfiehlt sich, ihn explizit anzugeben.

Mit `bytes.fromhex()` könnt ihr die Hexadezimalwerte in Bytes umwandeln:

```
>>> bytes.fromhex("c3 b6")
b'\xc3\xb6'
```

UTF-16 und UTF-32

Der Unterschied zwischen diesen und UTF-8 ist in der Praxis erheblich. Im Folgenden möchte ich euch nur kurz an einem Beispiel zeigen, dass hier eine eine Round-Trip-Konvertierung einfach fehlschlagen kann:

```
>>> 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
```

Die Kodierung von lateinischen Buchstaben in UTF-8 und die anschließende Dekodierung in UTF-16 führte zu einem Text, der auch Zeichen aus dem chinesischen, japanischen oder koreanischen Sprachraum sowie römische Ziffern enthält. Die Dekodierung desselben Byte-Objekts kann zu Ergebnissen führen, die nicht einmal in derselben Sprache sind oder gleich viele Zeichen enthalten.

18.2.4 Python 3 und Unicode

Python 3 setzt voll und ganz auf Unicode und speziell auf UTF-8:

- Der Quellcode von Python 3 wird standardmäßig in UTF-8 angenommen.
- Texte (str) sind standardmäßig Unicode. Kodierter Unicode-Text wird als Binärdaten (Bytes) dargestellt.
- Python 3 akzeptiert viele Unicode-Codepunkte in Bezeichnern.
- Pythons re-Modul verwendet standardmäßig das `re.UNICODE`-Flag und nicht `re.ASCII`. Das bedeutet, dass z.B. `r"\w"` auf Unicode-Wortzeichen passt, nicht nur auf ASCII-Buchstaben.
- Die Standardkodierung in `str.encode()` und `bytes.decode()` ist UTF-8.

Die einzige Ausnahme könnte `open()` sein, das plattformabhängig ist und daher vom Wert von `locale.getpreferredencoding()` abhängt:

```
>>> import locale
>>> locale.getpreferredencoding()
'UTF-8'
```

18.2.5 Built-In Python-Funktionen

Python verfügt über eine Reihe von eingebauten Funktionen, die sich in irgendeiner Weise auf Zeichenkodierungen beziehen:

`ascii()`, `bin()`, `hex()`, `oct()`
: geben einen String aus.

`bytes`, `str`, `int`
: sind Klassenkonstruktoren für ihre jeweiligen Typen, die die Eingabe in den gewünschten Typ konvertiert.

`ord()`, `chr()`
: sind insofern invers zueinander, als die Python-Funktion `ord()` ein `str`-Zeichen in seinen `base=10`-Codepunkt umwandelt, während `chr()` das Gegenteil tut.

Im Folgenden findet ihr einen detaillierteren Blick auf jede dieser neun Funktionen:

Funktion	Rückgabetyp	Beschreibung
`ascii()`	`str`	ASCII-Darstellung eines Objekts, wobei nicht-ASCII-Zeichen escaped werden
`bin()`	`str`	binäre Darstellung einer ganzen Zahl mit dem Präfix `0b`
`hex()`	`str`	hexadezimale Darstellung einer ganzen Zahl mit dem Präfix `0x`
`oct()`	`str`	Oktaldarstellung einer ganzen Zahl mit dem Präfix `0o`
`bytes`	`bytes`	konvertiert die Eingabe in bytes-Typ
`str`	`str`	konvertiert die Eingabe in str-Typ
`int`	`int`	konvertiert die Eingabe in `int`-Typ
`ord()`	`int`	konvertiert ein einzelnes Unicode-Zeichen in seinen Integer-Codepunkt
`chr()`	`str`	wandelt einen Integer-Codepunkt in ein einzelnes Unicode-Zeichen um

Sonderzeichen

A

B

C

D

E

F

G

H

I

K

L

M

P

Q

R

S

T

U

V

W