Python-Kurs.eu – Python-Tutorial

Eine lebende Buchfassung des Online-Tutorials

Bernd Klein

14. August 2026

Python-Kurs.eu – Python-Tutorial

Eine lebende Buchfassung des Online-Tutorials

Autor: Bernd Klein
Website: https://www.python-kurs.eu
Stand dieser Ausgabe: 14. August 2026

Diese Ausgabe wurde automatisiert aus den autoritativen Jupyter-Notebooks von `python-kurs.eu` erzeugt. Satz und technische Produktion erfolgen reproduzierbar mit Python, Pandoc, LuaLaTeX und - soweit erforderlich - PythonTeX.

Die Online-Fassung kann aktueller sein als diese PDF-Ausgabe. Trotz sorgfältiger Prüfung kann für die Fehlerfreiheit von Texten und Programmen keine Gewähr übernommen werden.

Die im Anhang beworbenen Verlagstitel sind eigenständige Bücher beim Carl Hanser Verlag. Die vorliegende automatisch erzeugte Ausgabe ist **keine Veröffentlichung des Carl Hanser Verlags**.

Das Python-Logo wird zur Kennzeichnung des Bezugs zur Programmiersprache Python verwendet. Python und das Python-Logo sind Marken der Python Software Foundation.

Inhaltsverzeichnis

1 Python3 Entstehung Python **1**
1.0.1 Geschichte 1

2 Python3 Interaktiv **3**
2.0.1 Der Interpreter, eine interaktive Shell 3

3 Python3 Skript Ausfuehren **8**
3.0.1 Ausführen von Python-Code 8

4 Python3 Bloecke **13**
4.0.1 Strukturierung durch Einrückung 13

5 Python3 Variablen **15**
5.0.1 Datentypen und Variablen 15

6 Python3 Operatoren **28**
6.0.1 Ausdrücke und Operatoren 28

7 Python3 Sequentielle Datentypen **30**
7.0.1 Sequentielle Datentypen 30

8 Python3 Listen **41**
8.0.1 Listen 41

9 Python3 Sets Mengen **47**
9.0.1 Mengen 47

10 Mengen Beispiel **58**
10.0.1 Python-Mengen: Umfangreiches Beispiel 58

11 Python3 Deep Copy **63**
11.0.1 Flache und tiefe Kopie 63

12 Python3 Dictionaries **69**
12.0.1 Dictionaries 69

13 Python3 Eingabe **86**
13.0.1 Eingabe 86

14 Python3 Bedingte Anweisungen **89**
14.0.1 Bedingte Anweisungen 89

15 Struktureller Musterabgleich **96**
15.0.1 Struktureller Musterabgleich 96

16 Abenteuer Spiel Mit Strukturellem Pattern Matching **101**
16.0.1 Ein Text-basiertes Abenteuerspiel . . . 101
16.0.2 Ein Text-Abenteuerspiel . . . 101
16.0.3 Übereinstimmung spezifischer Muster: . . . 102
16.0.4 Übereinstimmung mehrerer Werte: . . . 102
16.0.5 Verschiedene Muster mit gleichem Ergebnis . . . 104
16.0.6 Erfassen übereinstimmender Teilmuster . . . 104
16.0.7 Patterns mit Bedingungen . . . 104

17 Python3 Schleifen **105**
17.0.1 Schleifen . . . 105

18 Python3 For Schleife **115**
18.0.1 For-Schleifen . . . 115

19 Python3 Walross Operator **127**
19.0.1 Zuweisungsausdrücks alias Walross-Operator . . . 127

20 Zuweisungsausdrücke **132**
20.0.1 Zuweisungsausdrücke, auch bekannt als Walrus-Operator. . . . 132

21 Python3 Print **139**
21.0.1 print . . . 139

22 Python3 Formatierte Ausgabe **142**
22.0.1 Formatierte Ausgabe . . . 142

23 Arbeiten Mit Python Dictionaries **159**
23.0.1 Arbeiten mit Dictionaries . . . 159
23.1 Einführung . . . 159
23.1.1 Coffee, Dictionary and a Loop . . . 159

24 Python3 Funktionen **164**
24.0.1 Funktionen . . . 164

25 Python3 Parameter **181**
25.0.1 Parameterübergabe: Parameter und Argumente . . . 181

26 Python3 Namensraum **188**
26.0.1 Namensräume und Geltungsbereiche . . . 188

27 Python3 Global Lokal **190**
27.0.1 Globale und Lokale Variablen . . . 190

28 Python3 Dateien **196**
28.0.1 Dateien lesen und schreiben . . . 196

29 Python3 Modularisierung **202**
29.0.1 Modulare Programmierung und Module . . . 202

30 Python3 Pakete **214**
30.0.1 Pakete in Python . . . 214

31 Python3 Ausnahmebehandlung **220**
31.0.1 Fehler und Ausnahmen 220

Weitere Bücher von Bernd Klein **227**

1 Python3 Entstehung Python

1.0.1 Geschichte

Was hat die Geschichte von Python mit dem Alphabet zu tun? Beide beginnen mit ABC. Im Falle von Python die Programmiersprache ABC.

Die Sprache wurde Anfang der 1990er Jahre von Guido van Rossum am Zentrum für Mathematik (Centrum voor Wiskunde en Informatica) in Amsterdam entwickelt. Ursprünglich war sie als Nachfolger für die Lehrsprache ABC entwickelt worden und sollte auf dem verteilten Betriebssystem Amoeba laufen. Guido van Rossum hatte auch an der Entwicklung der Sprache ABC mitgewirkt, so dass seine Erfahrungen mit ABC auch in Python einflossen.

Auch wenn wir auf dieser Webseite nicht mit Python-Schlangen geizen, hat der Name der Programmiersprache Python nichts mit den Schlangen zu tun. Für Guido van Rossum stand vielmehr die britische Komikertruppe Monty Python mit ihrem legendären Flying Circus Pate für den Namen.

Guido van Rossum schrieb 1996 über die Entstehung des Namens seiner Programmiersprache: "Vor über sechs Jahren, im Dezember 1989, suchte ich nach einem 'Hobby'-Programmier-Projekt, dass mich über die Woche um Weihnachten beschäftigen konnte. Mein Büro … war zwar geschlossen, aber ich hatte einen PC und sonst nichts vor. Ich entschloss mich einen Interpreter für die neue Skripting-Sprache zu schreiben, über die ich in der letzten Zeit nachgedacht hatte: ein Abkömmling von ABC, der UNIX/C-Hackern gefallen würde. Python hatte ich als Arbeitstitel für das Projekt gewählt, weil ich in einer leicht respektlosen Stimmung war (und ein großer Fan von Monty Python's Flying Circus)."

Dennoch sind Assoziationen mit Schlangen möglich und sinnvoll: Man denke nur an das Python-Toolkit "Boa" oder die Programmiersprache Cobra. Außerdem ist eine Schlange im Python-Logo.

Guido van Rossum sagte in einem Interview: "Anfang der 80er Jahre habe ich an der CWI mit einem Team an der Sprache ABC gearbeitet. Ich hatte keine Ahnung wie ABC Python beeinflussen würde. Ich dachte an meine Erfahrungen und an den Frust mit ABC, und ich entschied mich eine einfache Skriptsprache zu entwerfen. Sie sollte die Vorteile von ABC haben, nicht aber die Probleme/Nachteile. Es entstand eine einfache virtuelle Maschine, ein einfacher Parser, eine einfache Laufzeitumgebung und eine Syntax, die Einrückungen für die Gruppierung von Ausdrücken verwendet, und ein paar Datentypen: Dictionaries, Listen, Strings und Numbers/Integer."

Das Zen von Python

```
Schön ist besser als hässlich.
Explizit ist besser als implizit.
Einfach ist besser als komplex.
Komplex ist besser als kompliziert.
Flach ist besser als verschachtelt.
```

```
Spärlich ist besser als dicht.
Lesbarkeit zählt.
Sonderfälle sind nicht speziell genug, um gegen die Regeln zu verstoßen.
Obwohl Praktikabilität die Reinheit übertrifft,
Fehler sollten niemals stillschweigend vergehen.
Sofern nicht ausdrücklich zum Schweigen gebracht.
Verweigern Sie angesichts von Zweideutigkeiten die Versuchung zu raten.
Es sollte einen - und vorzugsweise nur einen - offensichtlichen Weg geben, dies zu tun.
Obwohl dieser Weg zunächst vielleicht nicht offensichtlich ist, es sei denn, Sie sind Nied
Jetzt ist besser als nie.
Obwohl nie oft besser ist als * gerade * jetzt.
Wenn die Implementierung schwer zu erklären ist, ist es eine schlechte Idee.
Wenn die Implementierung leicht zu erklären ist, kann dies eine gute Idee sein.
Namespaces sind eine großartige Idee - lasst uns mehr davon machen!
```

Wie sich Python entwickelt hat

Guido van Rossum hat die erste Version von Python (v. 0.9.0) im Februar 1991 veröffentlicht. Darin waren bereits Ausnahmenbehandlung, Funktionen und die Kern-Datentypen enthalten. Python v. 0.9.0 war bereits objektorientiert und modular.

Python v. 1.0, veröffentlicht im Januar 1994, kam mit Werkzeugen für funktionale Programmierung wie lambda, map, filter und reduce.

Python v. 2.0 lieferte die Listen-Abstraktion (list comprehension), einen vollständigen Garbage Collector und unterstütze Unicode. Python v. 2.0 wurde im Oktober 2000 vorgestellt.

Python musste weitere 8 Jahre gedeihen, bevor die Version 3.0 veröffentlicht wurde. Python V. 3.0 ist auch bekannt unter den Namen “Python 3000” und “Py3K”. Ein Schwerpunkt in 3.0 lag auf der Beseitigung von redundanten Programmteilen um das 13te Gesetz der Zen of Python zu erfüllen: “Es sollte einen - und bevorzugt genau einen - offensichtlichen Weg geben, es zu tun.” Auf die wesentlichen Unterschiede können wir an dieser Stelle aber noch nicht eingehen.

Einige Änderungen in Python 3.0:

- Drucken ist jetzt eine Funktion
- Ansichten und Iteratoren anstelle von Listen
- Die Regeln für die Bestellung von Vergleichen wurden vereinfacht. Z.B. Eine heterogene Liste kann nicht sortiert werden, da alle Elemente einer Liste miteinander vergleichbar sein müssen.
- Es gibt nur noch einen ganzzahligen Typ, d. H. Int. lang ist auch int.
- Die Division von zwei ganzen Zahlen gibt einen Float anstelle einer ganzen Zahl zurück. “//” kann verwendet werden, um das “alte” Verhalten zu haben.
- Text Vs. Daten anstelle von Unicode Vs. 8 Bit

2 Python3 Interaktiv

2.0.1 Der Interpreter, eine interaktive Shell

Zu den Begriffen Interaktiv und Shell

Interaktive Shells sind wie Muschelgehäuse Interaktiv kommt aus dem Lateinischen “inter agere”. Das Verb “agere” bedeutet unter anderem “tun”, “treiben”, “betreiben”, “handeln”. “inter” bezeichnet die zeitliche und örtliche Position zu Dingen und Ereignissen, also “zwischen” zwei oder “inmitten” (oder “unter”) vielen Objekten, Personen oder Ereignissen. “inter agere” bedeutet also “dazwischen handeln”. In diesem Sinne steht die interaktive Shell zwischen dem Anwender und dem Betriebssystem (Linux, Unix, Windows oder anderen) bzw. dem zu interpretierenden Programm (z.B. Python). Die interaktive Shell steht aber auch zeitlich zwischen den einzelnen Aktionen, d.h. zwischen den einzelnen Befehlen und Kommandos. Der Benutzer gibt einen Befehl ein, die Shell führt diesen unverzüglich aus, liefert sofort das Ergebnis und wartet dann auf den nächsten Befehl.

In Englisch steht das Wort “shell” für eine Schale, einen Panzer oder ganz allgemein eine Hülle oder Schutzhülle. “Shell” bezeichnet auch ein Schneckenhaus und das Gehäuse, das eine Muschel umschließt. Ebenso liegt eine Shell auch zwischen einem Betriebssystem und dem Benutzer. Wie eine Muschelschale schützt sie zum einen das Betriebssystem vor dem Benutzer und gleichzeitig erspart sie dem Benutzer die Benutzung der “primitiven” und schwer verständlichen Basisfunktionen, indem sie ihm komfortable Befehle zur Kommunikation mit dem Computer zur Verfügung stellt.

Auch die Programmiersprache Python bietet dem Anwender eine komfortable Kommandozeilen-Schnittstelle, die sogenannte Python-Shell, die man manchmal auch als interaktive Python-Shell bezeichnet. Man könnte meinen, dass es sich bei dem Begriff “interaktive Shell” um eine Tautologie handelt, da ja, so wie wir es oben beschrieben haben, Shells immer interaktiv sind. Dies ist jedoch nicht so: Es gibt auch vor allem im Umfeld von Linux und Unix Shells, die als Subshell aufgerufen werden und nicht interaktiv ausgeführt werden.

Benutzung der Python-Shell

Die meisten Linux-Distributionen kommen mit einem vorinstallierten Python und damit auch mit einer interaktiven Python Shell. Der Interpreter befindet sich üblicherweise im Pfad /usr/bin/ (aber manchmal auch in /usr/local/bin).

Man kann Python im interaktiven Modus aufrufen, indem man den Interpreter in einer Linux-Shell (Bash-Shell) ohne Parameter aufruft. Also

Python meldet sich mit Informationen über die installierte Version:

Wenn man genauer auf die obige Ausgabe schaut, das heißt auf die Versionsnummer, erkennt man ein Problem. Wir wollten uns ja mit Python 3 beschäftigen, aber Python 2.7.6 ist installiert. In diesem Fall muss man gegebenenfalls Python3 nachinstallieren und anschließend mit python3 den interaktiven Interpreter starten:

Ein einfacher Weg zu prüfen, welche Version installiert ist, ist nach der Eingabe von "python" die TAB-Taste zu drücken. Es folgt eine Auflistung von weiteren Optionen und Versionen, falls vorhanden.

Der Interpreter steht nun mit dem sogenannten Eingabeprompt ("»>") für Eingaben bereit. Man kann jetzt beliebigen Python-Code eingeben, der nach dem Quittieren mit der Eingabetaste (Return-Taste) sofort ausgeführt wird. So kann man sehr schnell interaktiv ein Hello-World-Programm "schreiben":

```
print("Hello World")
```

```
Hello World
```

print verlangt übrigens im Gegensatz zu Python 2.x zwingend eine Klammer. print ist eine Funktion in Python3 und Funktionen werden mit Klammern geschrieben.

Der interaktive Interpreter erlaubt noch eine Bequemlichkeit, die innerhalb eines Python-Skriptes nicht zulässig ist. Man kann einfach print weglassen und nur den Ausdruck hinschreiben. Er wird dann automatisch ausgegeben:

```
"Hello World"
```

```
'Hello World'
```

Verlassen der Python-Shell

Wir haben zwar gerade erst damit begonnen in der Python-Shell "herumzuspielen", dennoch wollen sicherlich einige das Programm bereits verlassen. Ctrl-C, was vielen Linux- und Unix-Benutzern häufig als erstes einfällt, funktioniert nicht! Den Python-Interpreter kann man per Kontrollsequenz nur mit Ctrl-D wieder verlassen oder unter Windows mit Ctrl-Z. Alternativ dazu, kann man die Python-Shell mittels Eingabe der Funktion exit() verlassen. Die Klammern hinter exit sind zwingend notwendig ab Python 3.

Python-Shell als "Taschenrechner"

Die interaktive Python-Shell kann man auch als einen simplen "Taschenrechner" nutzen. Man tippt einfach den gewünschten Ausdruck nach dem Prompt ein, und betätigt die Eingabetaste.

```
4.567 * 8.323 * 17
```

```
646.189397
```

Python kennt, wie die meisten anderen Programmiersprachen, die Regel "Punktrechnung geht vor Strichrechnung", wie wir im folgenden Beispiel sehen können:

```
(1 + 42) *2
```

```
86
```

```
1 + 42 * 2
```

```
85
```

Der jeweils letzte Ausgabewert wird vom Interpreter in einer speziellen Variablen automatisch gespeichert. Der Name der Variable ist einfach ein Unterstrich (underscore), also "_". Das Ergebnis der letzten Berechnung kann man sich also wieder ausgeben lassen:

```
_
```

```
85
```

Der Unterstrich kann im Prinzip wie eine normale Variable benutzt werden:

```
_ * 3
```

```
255
```

Dies gilt allerdings nicht in einem Python-Skript oder Python-Programm!

Benutzung von Variablen

In der Python-Shell kann man auch ganz einfach Variablen benutzen. Wenn Sie genau wissen wollen, was es mit Variablen und Datentypen auf sich hat, empfehlen wir in unserem Kapitel über Datentypen und Variablen weiterzulesen. Das folgende Beispiel wendet sich an Benutzer, die bereits ein wenig Erfahrung mit Programmierung haben. Man kann also ganz einfach Werte in Variablen speichern. Variablennamen bedürfen keiner besonderen Kennzeichnung wie in anderen Sprachen, wie zum Beispiel das Dollarzeichen in Perl:

```
maximal = 124
breite = 94
print(maximal - breite)
```

```
30
```

Mehrzeilige Anweisungen

Bis jetzt haben wir noch nicht über mehrzeilige Anweisungen gesprochen. Anfänger werden hier vielleicht Probleme haben, da noch einige Grundlagen zum Verständnis fehlen, und sollten deshalb besser mit den folgenden Kapitel weitermachen.

Wir demonstrieren, wie die interaktive Shell mit mehrzeiligen Anweisungen, wie zum Beispiel For-Schleifen umgeht.

```
staedte = ["Freiburg", "Zürich", "Berlin", "Ulm", "München"]
for stadt in staedte:
    print(stadt)
```

```
Freiburg
```

```
Zürich
Berlin
Ulm
München
```

Achtung: Nachdem man die Zeile “for stadt in staedte:” eingetippt hat, erwartet die interaktive Shell im Folgenden eingerückte Zeilen. Dies teilt uns die Shell mit einem geänderten Prompt mit: Statt dem gewohnten “»>” erhalten wir nun eine Folge von drei Punkten “...”. Wir müssen also alle Anweisungen, die mit der Schleife ausgeführt werden sollen um die gleiche Anzahl von Leerzeichen einrücken, mindestens ein Leerzeichen! Wenn wir fertig sind, müssen wir nach der letzten Zeile noch einmal zusätzlich die Eingabetaste drücken.

```
for n in range(20,40,4):
     res = n ** 2
     print(res)
```

```
400
576
784
1024
1296
```

Strings - Zeichenketten

Strings sind nichts anderes, als eine sequentielle Abfolge von Zeichen. Um einen String zu definieren, werden Anführungszeichen verwendet. Wurde ein String einmal definiert, kann er zur Laufzeit nicht mehr geändert werden. Dieses Thema behandeln wir in einem anderen Kapitel.

```
"Hello" + " " + "World"
```

```
'Hello World'
```

Um Strings in mehreren Zeilen zu schreiben, braucht es die dreifachen Anführungszeichen:

```
stadt = """
    Berlin ist die Hauptstadt
    von Deutschland und ist
    gleichzeitig auch ein Bundesland.
    """
print(stadt)
```

```
    Berlin ist die Hauptstadt
    von Deutschland und ist
    gleichzeitig auch ein Bundesland.
```

Man kann Strings sogar multiplizieren. Dabei passiert nicht anderes, als eine Verkettung der Zeichenkette um den gegebenen Faktor.

```
".-." * 4
```

'.-..-..-..-.'

3 Python3 Skript Ausfuehren

3.0.1 Ausführen von Python-Code

Im letzten Kapitel haben wir ein wenig mit einfachen Python-Kommandos in der Python-Shell herumgespielt. Programme werden aber normalerweise nicht interaktiv eingetippt sondern in Dateien gespeichert. Wir werden nun unser erstes "richtiges" Python-Programm schreiben. Wir starten mit dem "obligatorischen" Hallo-Welt-Skript. In der interaktiven Shell sieht es so aus:

Aber wie wir bereits gesagt haben, wollen wir nun ein "richtiges" Programm schreiben, also in einer Datei. Wir schreiben eine leichte Variation des Hallo-Welt-Themas. Wir wollen den Text "Python lernen" ausgeben. Dazu müssen wir die entsprechende print-Zeile in eine Datei schreiben. Um unsere Datei zu editieren und abzuspeichern, benötigen wir einen Editor. Sowohl unter Linux als auch unter Windows stehen eine Vielzahl von Editoren und Entwicklungsumgebungen zur Entwicklung von Python-Programmen zur Verfügung. Unter Linux kann man beispielsweise den vi, vim, emacs, geany, gedit und eine Vielzahl von weiteren Editoren benutzen. Der emacs steht beispielsweise auch unter Windows zur Verfügung. Als Entwicklungsumgebung eignet sich sowohl unter Linux als auch unter Windows das quelloffene Programm eclipse hervorragend.

Wenn Sie sich also für einen Editor entschieden haben, geben Sie dort bitte die folgende Zeile ein:

Dann speichern Sie bitte das Mini-Skript unter *python_lernen.py*. Die Dateiendung "py" ist unter Linux für die Funktionsweise nicht unbedingt wichtig, solange man das Python-Programm nicht als Modul verwenden will. Die Endung sollte man aber dennoch verwenden, denn dadurch lassen sich Python-Dateien direkt am Namen erkennen. Wie bereits geschrieben, ist die Endung jedoch zwingend notwendig, wenn man die Datei als Modul verwenden will.

Starten eines Python-Skriptes

Nehmen wir an, dass unser Skript in einem Unterverzeichnis "python" des Benutzers monty gespeichert ist, dann kann man es ganz leicht in einer Shell starten:

Im Folgenden finden Sie die gleiche Eingabesequenz wie im Bild:

Starten unter Windows

Unter Windows kann man das Programm aus der Eingabeaufforderung heraus starten: (Start -> Alle Programme -> Zubehör -> Eingabeaufforderung):

Natürlich lässt sich ein Python-Skript oder Python-Programm unter Windows auch in gewohnter Manier durch Doppelklick auf das Icon starten:

Nach dem Doppelklick wird man jedoch bei unserem Beispiel eine unangenehme Über-

raschung erleben. Ein Ausgabefenster erscheint zwar, aber nur so kurz, dass man keine Ausgaben lesen kann. Üblicherweise löst man dieses Problem mit einem Trick: Man fügt hinter die letzte Programmzeile eine Eingabeaufforderung ein. Das Kommando “input()” wartet auf eine Eingabe. Das bedeutet, dass man die vorigen Ausgaben solange lesen kann, bis man eine Eingabe eingegeben hat, - also z.B. auch die leere Eingabe - und diese dann durch drücken der Return-Taste bestätigt hat.

Python-Interna

Höchstwahrscheinlich haben Sie irgendwo gelesen, dass die Python-Sprache eine interpretierte Programmier- oder Skriptsprache ist. Die Wahrheit ist: Python ist sowohl eine interpretierte als auch eine kompilierte Sprache. Aber Python als kompilierte Sprache zu bezeichnen, wäre irreführend. (Am Ende dieses Kapitels finden Sie die Definitionen für Compiler und Interpreter, falls Sie mit den Konzepten noch nicht vertraut sind!) Es wird davon ausgegangen, dass der Compiler den Python-Code in die Maschinensprache übersetzt. Python-Code wird in Zwischencode übersetzt, der von einer virtuellen Maschine ausgeführt werden muss, die als PVM (Python Virtual Machine) bezeichnet wird. Dies ist ein ähnlicher Ansatz wie bei Java. Es gibt sogar eine Möglichkeit, Python-Programme in Java-Bytecode für die Java Virtual Machine (JVM) zu übersetzen. Dies kann mit Jython erreicht werden.

Die Frage ist, muss ich meine Python-Skripte kompilieren, um sie schneller zu machen, oder wie kann ich sie kompilieren? Die Antwort ist einfach: Normalerweise müssen Sie nichts tun und sollten sich nicht darum kümmern, da “Python” bereits das Denken für Sie erledigt, d. H. Die erforderlichen Schritte automatisch ausführt.

Aus welchem Grund möchten Sie ein Python-Programm manuell kompilieren? Kein Problem. Dies kann mit dem Modul py_compile entweder über die Interpreter-Shell erfolgen:

Ausgabe::

oder verwenden Sie den folgenden Befehl an der Shell-Eingabeaufforderung

In beiden Fällen können Sie zwei Dinge bemerken: Erstens gibt es ein neues Unterverzeichnis “pycache”, falls es noch nicht vorhanden ist. In diesem Unterverzeichnis finden Sie eine Datei “my_first_simple_script.cpython-34.pyc”. Dies ist die kompilierte Version unserer Datei im Bytecode.

Sie können auch alle Python-Dateien automatisch mit dem Compileall-Modul kompilieren. Sie können dies über die Shell-Eingabeaufforderung tun, indem Sie compileall.py ausführen und den Pfad des Verzeichnisses angeben, das die zu kompilierenden Python-Dateien enthält:

Aber wie gesagt, Sie müssen und sollten sich nicht um das Kompilieren von Python-Code kümmern. Die Zusammenstellung ist aus gutem Grund dem Benutzer verborgen. Einige Neulinge fragen sich manchmal, woher diese ominösen Dateien mit dem Suffix .pyc stammen könnten. Wenn Python Schreibzugriff für das Verzeichnis hat, in dem sich das Python-Programm befindet, speichert es den kompilierten Bytecode in einer Datei, die mit dem Suffix .pyc endet. Wenn Python keinen Schreibzugriff hat, funktioniert das Programm trotzdem. Der Bytecode wird erzeugt, aber beim Beenden des Programms verworfen. Bei jedem Aufruf eines Python-Programms prüft Python, ob eine kompilierte Version mit dem Suffix .pyc vorhanden ist. Diese Datei muss neuer sein als die Datei mit dem Suffix .py. Wenn eine solche Datei vorhanden ist, lädt Python den Bytecode, wodurch die Startzeit des Skripts beschleunigt wird. Wenn keine Bytecode-Version vorhanden ist, erstellt Python den Bytecode, bevor die Ausführung des Programms gestartet wird. Die Ausführung eines Python-Programms bedeutet die Ausführung des Bytecodes auf dem Python.

Virtuelle Maschine (PVM).

Kompilierung eines Python-Skripts Jedes Mal, wenn ein Python-Skript ausgeführt wird, wird ein Bytecode erstellt. Wenn ein Python-Skript als Modul importiert wird, wird der Bytecode in der entsprechenden .pyc-Datei gespeichert. Im Folgenden wird also keine Bytecodedatei erstellt:

` monty @ python: ~ / python $ python my_first_simple_program.py Mein erstes einfaches Python-Skript! monty @ python: ~ / python $ ‘ Beim Import in die folgende Python2-Sitzung wird eine Bytecodedatei mit dem Namen “Mein_erstes_einfaches_Python_Programm.pyc” erstellt:

```
monty@python:~/tmp$ ls
my_first_simple_program.py
monty@python:~/tmp$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import my_first_simple_script
My first simple Python script!
>>> exit()
monty@python:~/tmp$ ls
my_first_simple_program.py  Mein_erstes_einfaches_Python_Programm.pyc
monty@python:~/tmp$
```

Ausführbare Skripte unter Linux

Dieses Kapitel kann von Windows-Benutzern übersprungen werden. Mach dir keine Sorgen! Es ist nicht wesentlich!

Bisher haben wir unsere Python-Skripte mit gestartet ` python3 meine_Datei.py ‘

auf der Bash-Befehlszeile. Ein Python-Skript kann auch wie jedes andere Skript unter Linux gestartet werden, z. Bash-Skripte. Zu diesem Zweck sind zwei Schritte erforderlich: Die Shebang-Zeile #! / usr / bin / env python3 muss als erste Zeile Ihrer Python-Codedatei hinzugefügt werden. Alternativ kann diese Zeile #! / Usr / bin / python3 sein, wenn dies der Speicherort Ihres Python-Interpreters ist. Anstatt env wie in der ersten Shebang-Zeile zu verwenden, wird der Interpreter zum Zeitpunkt der Ausführung des Skripts gesucht und gefunden. Dies macht das Skript portabler. Es hat jedoch auch das gleiche Problem: Der Pfad zu env kann auch pro Maschine unterschiedlich sein. Die Datei muss ausführbar gemacht werden: Der Befehl “chmod + x scriptname” muss auf einer Linux-Shell ausgeführt werden, z. Bash. “chmod 755 scriptname” kann auch verwendet werden, um Ihre Datei ausführbar zu machen. In unserem Beispiel:

` $ chmod + x Mein_erstes_einfaches_Python_Skript.py ‘

Wir veranschaulichen dies in einer Bash-Sitzung:

```
bernd@marvin:~$ more Mein_erstes_einfaches_Python_Skript.py
#!/usr/bin/env python3
print("Mein erstes einfaches Python-Skript!")
bernd@marvin:~$ ls -ltr my_first_simple_script.py
```

```
-rw-r--r-- 1 bernd bernd 63 Nov  4 21:17 my_first_simple_script.py
bernd@marvin:~$ chmod +x Mein_erstes_einfaches_Python_Skript.py
bernd@marvin:~$ ls -ltr my_first_simple_script.py
-rwxr-xr-x 1 bernd bernd 63 Nov  4 21:17 my_first_simple_script.py
bernd@marvin:~$ ./Mein_erstes_einfaches_Python_Skript.py
Mein erstes einfaches Python-Skript!
```

Compiler und Interpreter

Ein Compiler (auch Übersetzer genannt) ist ein Computerprogramm, das ein in einer Quellsprache, wie beispielsweise C oder C++, geschriebenes Programm - Quelle oder Quellprogramm genannt - in ein semantisch äquivalentes Programm (Zielprogramm) einer Zielsprache übersetzt. Üblicherweise handelt es sich dabei um die Übersetzung eines von einem Programmierer in einer Programmiersprache geschriebenen Quelltextes in Assemblersprache, Bytecode oder Maschinensprache. Das Übersetzen eines Quellprogramms in ein Zielprogramm durch einen Compiler wird auch als Kompilierung bezeichnet.

Ein Interpreter ist ein Programm, das einen Quellcode im Gegensatz zu Assemblern oder Compilern nicht direkt in ausführbaren Code, also eine ausführbare Datei, wandelt, sondern den Quellcode einliest, analysiert und ausführt. Die Analyse des Quellcodes erfolgt zur Laufzeit des Programms.

Hilfe Zu jedem Kommando und jeder Funktion kann man im Python-Interpreter auch interaktiv Hilfe anfordern.

```
help("print")
```

Ruft man help() ohne Parameter auf, gelangt man in den Help-Modus. Nun kann man sich mit “modules”, “keywords”, “symbols” oder “topics” ausführliche Informationen zu den verfügbaren Modulen, Schlüsselwörtern, Symbolen und Themen geben lassen.

```
help()
```

```
Welcome to Python 3.7's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

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
```

4 Python3 Bloecke

4.0.1 Strukturierung durch Einrückung

Blöcke

Ein Block ist eine Gruppe von Anweisungen in einem Programm oder einem Skript. Üblicherweise besteht er aus wenigstens einer Anweisung und Deklarationen für den Block, abhängig von der Skript- oder Programmiersprache. Eine Sprache, die die Strukturierung mit Blöcken ermöglicht, wird als strukturierte Programmiersprache bezeichnet. Im allgemeinen können Blöcke wieder Blöcke enthalten, d.h. wir erhalten verschachtelte Blockstrukturen. In einem Skript oder einem Programm dienen Blöcke auch dazu mehrere Anweisungen so zu gruppieren, dass sie wie eine Anweisung behandelt werden können. Außerdem werden Blöcke auch dazu genutzt, den Geltungsbereich von Variablen und Funktionen einzuschränken.

In einfachen Programmiersprachen wie Basic und Fortran gab es ursprünglich keine explizite Möglichkeit Blockstrukturen zu nutzen. Programmierer mussten sich auf "goto"-Konstrukte beschränken. Heutzutage sind "goto"s in der Programmierung verpönt, weil "Go to"-Programme zu Spaghetti-Code tendieren, d.h. die Kontrollstrukturen sind verwoben und verwickelt und meistens dadurch in ihrer Struktur und ihrem Zusammenspiel nur schwer zu durchschauen.

Zum ersten Mal wurden Blockstrukturen in ALGOL verwendet. Sie wurden als "compound statement" bezeichnet.

Programmier- und Skriptsprachen benutzen verschiedene Methoden, Anweisungen zu Blöcken zusammenzufassen:

- begin … end ALGOL, Pascal und andere Ein Codefragment in Pascal, was die Benutzung von Blöcken in dieser Programmiersprache zeigt:
- do … done and if… fi z.B. Bourne- und Bash-Shell
- Geschweifte Klammern: { … } In den meisten Programmier- und Skriptsprachen werden wohl geschweifte Klammern zur Strukturierung verwendet, so wie in C, C++, Perl und Java. Im folgenden Beispiel sehen wir eine bedingte Anweisung in C:
  Man beachte, dass die Einrückungen in dem obigen C-Programm zur Kompilierung nicht notwendig sind. Man könnte den Code - gegen alle Programmierästhetik - auch wie folgt schreiben:
  Man sollte sich dieses Beispiel gut merken, um die Vorteile von Python, über die wir im folgenden sprechen werden, besser zu verstehen!

Python-Blöcke durch Einrückungen

Das Strukturierungsprinzip von Python unterscheidet sich deutlich von anderen Programmiersprachen. Wie eingangs bereits beschrieben, strukturieren andere Programmiersprachen ihre Programmblöcke durch Schlüsselwörter, wie beispielsweise "begin", "end", "do", "done" oder geschweifte Klammern. Leerzeichen, Folgen von Leerzeichen oder Einrückungen sind

für die Compiler und Interpreter von den meisten Programmiersprachen ohne jede Semantik, d.h. sie werden überlesen. Dennoch wird Programmierern aber immer empfohlen, Blöcke durch gleichmäßge Einrückungen für menschliche Benutzer besser verständlich zu machen. In Python ist dies nun gänzlich anders. Hier haben Leerzeichen eine Bedeutung. Die Einrückung von Zeilen und die Benutzung von Leerzeichen am Anfang von Zeilen dienen hier als Strukturierungselement, so dass Programmierer "gezwungen" werden übersichtlichen Code zu schreiben, wenn sie ein lauffähiges Programm unter Python entwickeln wollen. Ein häufiges Strukturierungselement sind Anweisungen, die sich aus einem Anweisungskopf und einem Anweisungskörper zusammensetzen, wie z.B. die while- und die for-Schleife.

Wesentlich sind hierbei der Doppelpunkt am Ende des Anweisungskopfes und die gleichmäßige Einrückung der zugehörigen Anweisungen.

Im Folgenden präsentieren wir ein vollständiges Python-Programm, damit wir ein konkretes Beispiel mit Blöcken sehen können. Anfänger werden dieses Programm noch nicht verstehen können. Wir möchten Ihnen lediglich ein Beispiel eines strukturierten Programmes zeigen. Übrigens handelt es sich bei dem Programm um einen Algorithmus der Pythagoreische Tripel berechnet. Sie finden eine Erklärung der Pythagoreische Tripel in unserem Kapitel über for-Schleifen.

from math import sqrt n = input("Maximal Number?") n = int(n)+1 for a in range(1,n): for b in range(a,n): c_square = a**2 + b**2 c = int(sqrt(c_square)) if ((c_square - c**2) == 0): print(a, b, c)

Es gibt einen weiteren Aspekt der Strukturierung in Python, den wir bisher nicht erwähnt haben und den Sie im Beispiel sehen können. Schleifen und bedingte Anweisungen enden mit einem Doppelpunkt ":" - das Gleiche gilt für Funktionen und andere Strukturen, die Blöcke einführen. Alles in allem Python-Strukturen durch Doppelpunkte und Einrückungen.

5 Python3 Variablen

5.0.1 Datentypen und Variablen

Einführung

Haben Sie bereits in Sprachen wie C und C++ programmiert? Deswegen glauben Sie, dass Sie bereits genug über Datentypen und Variablen wissen? Sie wissen sicherlich viel, das stimmt, aber nicht genug, wenn es um Python geht. Deshalb lohnt es sich auf jeden Fall hier weiterzulesen.

Es gibt gravierende Unterschiede in der Art wie Python und C bzw. C++ Variablen behandeln. Vertraute Datentypen wie Ganzzahlen (Integer), Fließkommazahlen (floating point numbers) und Strings sind zwar in Python vorhanden, aber auch hier gibt es wesentliche Unterschiede zu C und C++. Dieses Kapitel ist also sowohl für Anfänger als auch für fortgeschrittene Programmierer zu empfehlen.

Benutzung von Variablen

Benötigt man in einem Python-Programm beispielsweise den Wert 42, so kann man sich diesen mittels einer Variablen "speichern". Dies geschieht mit einer Zuweisung:

```
i = 42
```

Diese Anweisung bezeichnet man als Zuweisung oder auch als Definition einer Variablen. Das Gleichheitszeichen darf man nicht als mathematisches Gleichheitszeichen sehen, sondern als "der Variablen i wird der Wert 42 zugewiesen", d.h. die Variable i zeigt nach der Zuweisung auf den Wert 42.

Wir können nun diese Variable zum Beispiel benutzen, indem wir ihren Wert mit print ausgeben lassen:

```
i = 42
print("Wert von i: ", i)
```

```
Wert von i:  42
```

Wir können sie aber auch auf der rechten Seite einer Zuweisung in einem Rechenausdruck verwenden, dessen Ergebnis dann einer Variablen zugewiesen wird:

```
x = i / 3.0 + 5.8
print(x)
```

```
19.8
```

Man kann eine Variable auch gleichzeitig auf der rechten und der linken Seite einer Zuweisung benutzen. Im Folgenden erhöhen wir den Wert der Variablen i um 1:

```
i = 1
i = i + 1
print(i)
```

```
2
```

Für obige Schreibweise wird üblicherweise die sogenannte erweiterte Zuweisung verwendet:

```
i = 1
i += 1
print(i)
```

```
2
```

Bei unveränderlichen Zahlen liefern `i = i + 1` und `i += 1` denselben Zahlenwert. Allgemein sind normale und erweiterte Zuweisungen aber **nicht vollständig gleichbedeutend**. Eine erweiterte Zuweisung wertet das Ziel nur einmal aus und kann bei veränderbaren Objekten eine In-place-Operation verwenden. So verändert `liste += andere_liste` normalerweise die vorhandene Liste, während `liste = liste + andere_liste` eine neue Liste erzeugt und den Namen anschließend daran bindet.

Erweiterte Zuweisungen gibt es für viele Operatoren, unter anderem `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `&=`, `|=` und `^=`.

In Python hat ein Variablenname selbst keinen fest eingebauten Datentyp. Ein Name kann während der Programmausführung nacheinander an Objekte verschiedener Typen gebunden werden. Präziser ist deshalb die Formulierung: **Namen referenzieren Objekte; die Objekte besitzen einen Typ.**

```
i = 42          # Typ wird implizit auf Integer gesetzt
i = 41 + 0.11   # Typ wird in float geändert
i = "fünfzig"   # Jetzt ist es ein string
```

Regeln zur Benennung von Variablen

Variablennamen sind in Python *Bezeichner* (Identifiers). Ein gültiger Bezeichner ist eine nicht leere Zeichenfolge. Vereinfacht gelten folgende Regeln:

- Das erste Zeichen darf ein Unterstrich `_` oder ein buchstabenartiges Unicode-Zeichen sein.
- Danach sind zusätzlich Ziffern erlaubt.
- Groß- und Kleinschreibung werden unterschieden: `wert`, `Wert` und `WERT` sind verschiedene Namen.
- Reservierte Python-Schlüsselwörter wie `if`, `while` oder `class` dürfen nicht als normale Bezeichner verwendet werden.
- Python erlaubt viele Nicht-ASCII-Schriften in Bezeichnern. Nicht jedes beliebige Unicode-Zeichen ist jedoch zulässig; insbesondere sind Symbole und Emoji keine normalen Bezeichnerzeichen.

Beispiele mit gültigen Variablennamen:

```
speed = 123.4
      = 89        # Greek
   = 10             # Hindi: pronounced gati
hız = 12             # Turkish
```

```
speed,      ,    , hız
```

```
(123.4, 89, 10, 12)
```

Namenskonventionen für Variablen

Bei Namen aus mehreren Wörtern hat sich in Python die Schreibweise mit Kleinbuchstaben und Unterstrichen etabliert, zum Beispiel `maximum_height`. Der offizielle Style Guide PEP 8 empfiehlt diese *snake_case*-Schreibweise für Funktionen und Variablen. Klassennamen werden dagegen üblicherweise in `CapWords`-Schreibweise geschrieben.

Solche Konventionen sind keine Syntaxregeln: Ein Programm läuft auch mit anders geschriebenen gültigen Namen. Einheitliche Namen machen Code jedoch deutlich leichter lesbar.

Variablen in anderen Sprachen

Dieses Unterkapitel ist speziell für C, C++ und Java-Programmierer gedacht, weil diese Programmiersprachen Standarddatentypen anders behandeln als Python. Diejenigen, die Python als erste Programmiersprache lernen, können gerne zum nächsten Unterkapitel springen.

In den meisten Programmiersprachen, wie z.B. C, ist es so, dass eine Variable einen festen Speicherplatz bezeichnet, in dem Werte eines bestimmten Datentyps abgelegt werden können. Während des Programmlaufes kann sich der Wert der Variable ändern, aber die Wertänderungen müssen vom gleichen Typ sein. Also kann man nicht in einer Variablen zu einem bestimmten Zeitpunkt eine Integerzahl gespeichert haben und dann diesen Wert durch eine Fließkommazahl überschreiben. Ebenso ist der Speicherort der Variablen während des gesamten Laufes konstant, kann also nicht mehr geändert werden. In Sprachen wie C und C++ wird der Speicherort bereits durch den Compiler fixiert.

Was wir bisher gesagt haben, trifft auf Sprachen wie C, C++ und Java zu. In diesen Sprachen müssen Variablen auch deklariert werden, bevor sie benutzt werden können.

Solche Deklarationen stellen sicher, dass das Programm Speicher für zwei Variablen mit den Namen x und y reserviert. Die Variablen Namen stehen für den Speicherort. Es ist so wie die beiden leeren Schuhkartons im Bild auf der rechten Seite. Diese beiden Schuhkartons sind mit x und y etikettiert. So wie die Schuhkartons ist auch der Speicher für x und y leer.

Will man Werte im Speicher ablegen, so kann man dies mit Zuweisungen verwirklichen. Die Art, wie man Werte zuweist, ist in den meisten Programmiersprachen gleich. Meistens wird das Gleichheitszeichen benutzt. Der Wert auf der rechten Seite wird in der Variablen auf der linken Seite gespeichert. Wir werden im folgenden C-Beispiel an beide vorher deklarierte Variablen den Wert 42 zuweisen. Das Ergebnis sehen wir auch im Bild veranschaulicht.

Es ist nicht schwer zu verstehen, was passiert, wenn wir einen neuen Wert einer der beiden Variablen zuweisen, zum Beispiel den Wert 78 der Variablen y.

Der Wert der Speicherstelle von y wurde ausgetauscht.

Wir haben nun gesehen, dass in Programmiersprachen wie C, C++ oder Java jede Variable einen eindeutigen Datentyp hat oder besser haben muss. Das bedeutet, dass falls beispielsweise eine Variable vom Typ Integer ist, sie nur Integer-Werte aufnehmen kann. In diesen Programmiersprachen müssen Variablen auch vor ihrer Benutzung deklariert werden. Deklaration bedeutet Bindung an einen Datentyp, der dann für den gesamten Programmablauf unveränderlich ist.

Variablen in Python

In Python haben wir eine gänzlich andere Situation. Zunächst einmal bezeichnen Variablen in Python keinen bestimmten Typ und deshalb benötigt man auch in Python keine Typdeklaration.

Eingangs haben wir bereits gesehen, dass Variablen durch Zuweisungen entstehen oder verändert werden:

```
i = 42
```

Variablen sind Objekt-Referenzen Wir wollen einen genaueren Blick auf die Implementierung von Variablen in Python werfen. Variablen referenzieren Objekte in Python. Die eigentlichen Daten sind jedoch im Objekt enthalten.

Da Variablen Objekte referenzieren und Objekte einen beliebigen Datentyp haben können, ist es nicht möglich, Variablen mit Datentypen zu verknüpfen. Dies ist ein riesiger Unterschied zu C, C++ oder Java, wo Variablen fest mit einem Datentyp assoziiert sind. Diese Verknüpfung gilt solange das Programm läuft.

Aus diesem Grund ist es möglich in Python Code wie den folgenden zu schreiben:

```
x = 42
print(x)
```

```
42
```

```
x = "Jetzt referenziert x einen String"
print(x)
```

```
Jetzt referenziert x einen String
```

Nun wollen wir etwas anderes aufzeigen. Betrachten wir den folgenden Code, in dem wir ein Integer-Objekt 42 anlegen und dieses der Variablen x zuweisen. Danach weisen wir x der Variablen y zu. Dies führt dazu, dass nun beide Variablen dasselbe Objekt referenzieren:

Was wird passieren, wenn wir nun die Zuweisung y = 78 ausführen? Python wird zuerst ein neues Integer-Objekt mit dem Inhalt 78 erzeugen und dann wird die Referenz von y auf dieses Objekt geändert, wie wir im folgenden Bild sehen können:

Höchst wahrscheinlich würden wir noch weitere Änderungen an den Variablen im Lauf

unseres Programmes erfahren. Zum Beispiel könnte der Variablen x ein String zugewiesen werden. Das Integer-Objekt “42” wird damit verwaist. Es muss von Python entfernt werden, da es von keiner anderen Variablen referenziert wird.

id-Funktion

Es stellt sich die Frage, wie man das oben gesagte überprüfen kann. Dazu bietet sich die Identitätsfunktion id() an. Die Identität einer Instanz dient dazu, sie von allen anderen Instanzen zu unterscheiden. Die Identität ist eine Ganzzahl, und sie ist innerhalb eines Programmes eindeutig. Die Identitätsfunktion id() liefert die Identität. So kann man prüfen, ob es sich um eine bestimmte Instanz handelt und nicht nur um eine mit dem gleichen Wert und Typ. Wir geben nochmals das obige Beispiel ein, lassen uns aber jeweils die Identität ausgeben:

```
x = 42
id(x)
```

```
94676682082336
```

```
y = x
id(x), id(y)
```

```
(94676682082336, 94676682082336)
```

```
y = 78
id(x), id(y)
```

```
(94676682082336, 94676682083488)
```

Wir stellen fest, dass sich die Identität erst ändert, nachdem wir y einen neuen Wert zugewiesen haben. Die Identität von x bleibt gleich, d.h. der Speicherort von x wird nicht geändert.

Python-Schlüsselwörter

Reservierte Schlüsselwörter dürfen nicht als normale Bezeichner verwendet werden. Statt eine möglicherweise veraltende Liste in den Quelltext zu schreiben, kann Python die aktuell gültigen Wörter selbst anzeigen:

```
import keyword

print(keyword.kwlist)       # reservierte Schlüsselwörter
print(keyword.softkwlist)   # kontextabhängige "soft keywords"
```

Zu den *soft keywords* gehören in aktuellen Python-Versionen beispielsweise `match`, `case`, `_` (im Pattern Matching) und `type` (in einer `type`-Anweisung). Außerhalb ihrer jeweiligen syntaktischen Kontexte können solche Wörter weiterhin normale Namen sein.

Verändern von Datentypen und Bindungen

Programmieren bedeutet Verarbeitung von Daten. Daten werden in einem Python-Programm durch Objekte repräsentiert. Diese Objekte können in Python

- eingebaut (built-in) sein, d. h. sie gehören zum Standardsprachumfang,
- aus Modulen stammen oder
- in der Anwendung selbst erzeugt werden.

Es gibt verschiedene Arten von Objekten für verschiedene Datentypen. Schauen wir uns einige wichtige eingebaute Datentypen an.

Zahlen

Python kennt die folgenden eingebauten (built-in) Datentypen für Zahlen:

- Ganzzahl (Integer) ~Normale Integer

```
x = 4321
```

~Eine Oktalzahl markiert man, indem man ein "0o", d.h. Ziffer Null gefolgt von einem kl

```
x = 0o11
print(x)
```

```
9
```

~Eine Hexadezimalzahl (Hexzahl) markiert man mit einem vorangestellten "0x" oder "0X":

```
x = 0x1A
print(x)
```

```
26
```

~Literale als Binärzahl lassen sich ähnlich leicht darstellen. Die führende Null wird r

```
x = 0b101010
x
```

```
42
```

Mit den Funktionen hex, bin, oct kann man eine Integer-Zahl in einen String wandeln, der der Stringdarstellung der Zahl in der entsprechenden Basis entspricht:

```
x = hex(19)
x
```

```
'0x13'
```

```
type(x)
```

```
str
```

```
x = bin(65)
x
```

```
'0b1000001'
```

```
x = oct(65)
x
```

```
'0o101'
```

```
oct(0b101101)
```

```
'0o55'
```

Python 3 besitzt für ganze Zahlen den Typ `int`. Seine Werte sind nicht auf die bei C typischen 32 oder 64 Bit beschränkt; solange genügend Speicher vorhanden ist, können ganze Zahlen sehr groß werden. (Python 2 unterschied historisch zwischen `int` und `long`; diese Unterscheidung spielt in Python 3 keine Rolle mehr.)

```
x = 787366098712738903245678234782358292837498729182728
print(x)
```

```
787366098712738903245678234782358292837498729182728
```

```
x * x * x
```

```
488123970070638215986770162105731315538827586091948617997871122950228891123960901
↪ 91918308618286311523282239313708275589787123005317148968569797875581092352
```

- Fließkommazahlen (`float`), zum Beispiel `3.14159` oder `17.3e+02`
- komplexe Zahlen (`complex`), zum Beispiel `1.2 + 3j`

Division und Floor Division

In Python 3 führt `/` eine *true division* aus. Bei zwei Integer-Operanden entsteht dabei ein `float`:

```
10 / 3
```

```
3.3333333333333335
```

Das Ergebnis von `10 / 3` ist also eine Fließkommazahl. Für eine Division mit mathematischem Abrunden verwendet Python den Operator `//` (*floor division*).

Wichtig: *floor* bedeutet Abrunden in Richtung **minus unendlich**, nicht einfach Abschneiden

der Nachkommastellen. Deshalb gilt beispielsweise `-10 // 3 == -4`, während `int(-10 / 3) == -3` ist.

```
x = 11
y = 3
z = x / y
type(z)
```

```
float
```

```
print(z)
```

```
3.6666666666666665
```

Mit `//` erhält man die Floor Division. Sind beide Operanden `int`, ist auch das Ergebnis ein `int`; ist mindestens ein Operand ein `float`, ist das Ergebnis ein `float`.

```
z = 10 // 3
type(z)
```

```
int
```

```
print(z)
```

```
3
```

String-Interna

Die Aufgabe der Computer der ersten Generation in den Vierziger- und Fünfzigerjahren des vorigen Jahrhunderts bestand - wegen technischer Restriktionen - im Wesentlichen in der Lösung numerischer Probleme. Textverarbeitung war zu dieser Zeit im Wesentlichen nur ein Traum. Heutzutage beschäftigen sich Computer in einem großen Umfang mit Textverarbeitung; die bedeutendsten Anwendungen stellen wohl die Suchmaschinen, allen voran Google, dar. Um mit Texten umzugehen, benötigen Programmiersprachen geeignete Datentypen. So werden Strings in nahezu allen modernen Programmiersprachen genutzt, um textuelle Informationen zu verarbeiten. Logisch gesehen ist ein String - wie Text - eine Folge von Zeichen. Die Frage bleibt, was ein Zeichen (englisch character) darstellt. In einem Buch oder auch einem Text, wie beispielsweise dem, den Sie gerade lesen, werden die Zeichen grafisch dargestellt, sogenannte Grapheme, die aus Linien und Kurven bestehen, die in bestimmten Verhältnissen angeordnet sind, sich überschneiden und so weiter.

In der Computertechnik stellt ein Character eine Informationseinheit dar. Die Zeichen (englisch characters) entsprechen Graphemen, den zugrundeliegenden Einheiten der geschriebenen und gedruckten Sprache. Vor Unicode gab es nur eine Eins-zu-eins-Beziehung zwischen Zeichen und Bytes, d.h., jedes einzelne Zeichen wurde durch ein Byte im Computer repräsentiert. Ein solches Byte repräsentiert dann das logische Konzept des Zeichens und steht für die gesamte Klasse der Grapheme dieses Zeichens. In dem Bild auf der rechten Seite zeigen wir verschiedene grafische Repräsentationen des Buchstabens “A”, also “A” in verschiedenen Fonts. Wir sehen, dass es im Druck verschiedene grafische Darstellungen des abstrakten Konzeptes “A” gibt. (Übrigens geht unser heutiges A auf eine ägyptische Hieroglyphe zurück, die einen Ochsenkopf symbolisiert) Alle grafischen Darstellungen haben

bestimmte Eigenschaften gemeinsam. Ein großes “A” wird in ASCII mit dem Byte-Wert 65 kodiert. Die meisten auf ASCII beruhenden Zeichenkodierungen, wie beispielsweise ISO-8859, sind allerdings auf 256 Bytes oder Zeichen beschränkt. ASCII selbst ist sogar auf 128 Zeichen beschränkt. Das ist natürlich genug für Sprachen wie Englisch, Deutsch oder Französisch, aber bei weitem nicht genug für Schriftsysteme, wie sie das Chinesische oder das Japanische erfordern. Wegen dieser Probleme wurde Unicode ins Leben gerufen. Unicode ist ein Standard, der entworfen worden ist, um jedes beliebige Zeichen von jeder Schrift darstellen zu können. Verschiedene Schriftsysteme können damit auch gleichzeitig verwendet werden, so kann ein Text in lateinischer Schrift beispielsweise mit kyrillischem, arabischen oder sogar chinesischem Text gemischt werden.

Jeder Buchstabe bzw. Zeichen wird in Unicode als eine 4-Byte große Zahl dargestellt, so entspricht beispielsweise das große “A” dem Wert U+0041. Während der erweiterte ASCII-Code auf 256 Zeichen pro Zeichensatz beschränkt ist, gibt es bei Unicode praktisch gesehen keinerlei Einschränkungen, denn mit 4 Bytes kann man 2564 Zeichen (also mehr als 4,29 Milliarden) abbilden. (Wegen der gewählten Zuordnung gibt es allerdings “nur” 1.112.064 mögliche Zeichen) Nur wenig mehr als 100.000 Zeichen also weniger als 1% des theoretisch möglichen Zeichenraumes sind in Unicode belegt. Aber die Frage bleibt, wie man Unicode implementiert. Wir könnten natürlich 4 Bytes pro Zeichen verwenden und damit den Standard Eins-zu-eins umsetzen. Aber das wäre eine Speichervergeudung. Ein Textdokument würde dann viermal so viel Speicherplatz benötigen wie bisher unter ASCII.

Es gibt verschiedene Zeichencodierungen für Unicode:

Name

Beschreibung

UTF-32

Hierbei handelt es sich um eine Eins-zu-eins Abbildung, d.h. jedes Unicode-Zeichen, was ja einer 4-Byte großen Zahl entspricht, wird auch in 4 Bytes abgespeichert. Ein Vorteil dieser Kodierung besteht darin, dass das n-te Zeichen in einem String in linearer Zeit berechnet (gefunden) werden kann, weil das n-te Zeichen immer an der Position 4×N vom Stringanfang aus gesehen beginnt. Der hohe Speicherverbrauch dieser Kodierung ist allerdings ein Problem.

UTF-16

Kaum jemand benötigt mehr als 65535, daher ist UTF-16, welches nur 2 Bytes benötigt, eine effizientere Alternative zu UTF-32. Ein Problem, dass sowohl UTF-32 und UTF-16 betrifft besteht darin, dass die Byte-Reihenfolge eines Zeichens vom Betriebssystem abhängt.

UTF-8

UTF8 ist eine Kodierung für Unicode mit variabler Länge, d.h. verschiedene Zeichen können verschiedene Kodierungslängen haben. So benötigen ASCII-Zeichen nur ein Byte pro Zeichen, die sogar dem original ASCII-Zeichensatz für die ersten 128 Zeichen entsprechen. Das bedeutet, dass diese Zeichen zwischen UTF-8 und ASCII ununterscheidbar sind. Aber die sogenannten “Erweiterten Lateinischen” Zeichen, denen beispielsweise die Umlaute wie ä, ö, ü angehören, benötigen zwei Bytes. Chinesische Zeichen benötigen drei und nur ganz spezielle extrem selten benutzte Spezialzeichen benötigen 4 Bytes in UTF-8. Ein Nachteil dieser Methode besteht darin, das n-te Zeichen eines Strings zu bestimmen. Die Berechnungszeit ist dann nicht mehr linear.

Benutzung von Strings

Ein String, oder Zeichenkette, kann man als eine Sequenz von einzelnen Zeichen sehen.

Jedes einzelne Zeichen eines Strings, kann über einen Index angesprochen werden. Im folgenden Beispiel sehen wir, wie der im Bild dargestellte String in Python definiert wird und wie wir auf ihn zugreifen können:

```
s = "Python"
print(s[0])
```

```
P
```

```
print(s[3])
```

```
h
```

Die Länge eines Strings kann man mit der Funktion len() bestimmen und damit kann man auch einfach beispielsweise auf das letzte oder vorletzte Zeichen eines Strings zugreifen:

```
s = "Python"
index_last_char = len(s) - 1
print(s[index_last_char])
```

```
n
```

```
print(s[index_last_char - 1])
```

```
o
```

Da es bei der praktischen Arbeit sehr häufig vorkommt, dass man auf einzelne Zeichen eines Strings von hinten zugreifen muss, wäre es sehr lästig, wenn man dies immer über den Umweg durch den Aufruf der Funktion len() machen müsste. Python bietet deshalb eine elegantere Möglichkeit. Die Indices werden auch von rechts durch negative Indices nummeriert, d.h. das letzte Zeichen wird mittels des Index -1, das vorletzte mittels -2 usw. angesprochen. Wir sehen dies in der folgenden Abbildung veranschaulicht:

Im Code sieht das wie folgt aus:

```
s = "Python"
last_character = s[-1]
print(last_character)
```

```
n
```

Strings können unter Benutzung von

- einzelnen Anführungszeichen (‘) ’Dies ist ein String mit einfachen Quotes’
- doppelten Anführungszeichen (“)”Mayers’ Dackel heißt Waldi”
- dreifachen Anführungszeichen (’ ’‘) oder (“ ““) ’‘’String in dreifachen Anführungszeichen können auch über mehrere Zeilen gehen und ’einfache’ und”doppelte” Anf.zeichen enthalten.” ’

angegeben werden.

Wichtige Stringfunktionen

Ein paar String-Funktionen:

- Konkatenation (englisch: Concatenation) Diese Funktion dient dazu mittels des "+"-Operators zwei Strings zu einem neuen String zusammenzuhängen: "Hello" + "World" -> "HelloWorld"
- Wiederholung (englisch: Repetition) Ein String kann wiederholt konkateniert werden. Dazu benutzt man den "*"-Operator. Beispiel: "abc" $*$ 3 wird zu "abcabcabc" oder "." $*$ 5 wird zu "...."
- Indexing "Python"[0] -> "P"
- Slicing Das englische Verb "to slice" bedeutet in Deutsch "schneiden" oder auch "in Scheiben schneiden". Letztere Bedeutung entspricht auch der Funktion von Slicing in Python. Man schneidet sich gewissermaßen eine "Scheibe" aus einem String heraus. [2:4] bedeutet im folgenden Ausdruck, dass wir aus dem String "Python" einen Teilstring herausschneiden, der mit dem Zeichen des Index 2 (inklusive) beginnt und bis zum Index 4 (exklusive) geht: "Python"[2:4] -> "th"

- Länge eines Strings len("Python") -> 6

Unveränderliche Zeichenketten

Wie in Java aber nicht wie in C oder C++, können Strings in Python nicht verändert werden. Versucht man eine indizierte Position zu ändern, erzeugt man eine Fehlermeldung:

```
s = "Some things are immutable!"
s[-1] = "."
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-61f4ed17a688> in <module>
      1 s = "Some things are immutable!"
----> 2 s[-1] = "."

TypeError: 'str' object does not support item assignment
```

Gleichheit und Identität bei Strings

Bei Strings muss man zwischen **Wertgleichheit** und **Objektidentität** unterscheiden:

- `a == b` prüft, ob beide Strings denselben Inhalt haben.
- `a is b` prüft, ob beide Namen exakt dasselbe Objekt referenzieren.

Für Textvergleiche verwendet man deshalb `==`, **nicht** `is`. Python-Implementierungen dürfen unveränderliche Objekte wie bestimmte String-Literale intern wiederverwenden (*interning*). Deshalb kann `a is b` bei zwei gleich aussehenden Strings je nach Entstehung der Objekte und Implementierung überraschend `True` oder `False` liefern. Darauf darf ein Programm nicht angewiesen sein.

`is` wird typischerweise für Identitätsprüfungen verwendet, besonders bei Singleton-Objekten wie `None`: `wert is None`.

```
a = "Linux"
b = "".join(["Li", "nux"])
print(a == b)   # gleicher Inhalt
print(a is b)   # normalerweise verschiedene Objekte
```

```
True
False
```

Escape- oder Fluchtzeichen

Es gibt Zeichenfolgen, die den Textfluss steuern, wie zum Beispiel ein Newline (Zeilenvorschub) oder Tabulator. Sie lassen sich nicht auf dem Bildschirm als einzelne Zeichen darstellen. Die Darstellung solcher Zeichen innerhalb von String-Literalen erfolgt mittels spezieller Zeichenfolgen, sogenannter Escape-Sequenzen. Eine Escape-Sequenz wird von einem Backslash eingeleitet, gefolgt von der Kennung des gewünschten Sonderzeichens. Übersicht der Escape-Zeichen:

- Zeilenfortsetzung
- \ Rückwärtsschrägstrich
- ' Einzelnes Anführungszeichen
- ”Doppeltes Anführungszeichen
- \a Glocke
- \b Rückschritt
- \e Ausmaskieren
- \0 Null
- \n Zeilenvorschub (linefeed, LF)
- \v Vertikaler Tabulator
- \t Horizontaler Tabulator
- \r Wagenrücklauf (carriage return, CR)
- \f Seitenvorschub
- \0XX Oktaler Wert
- \xXX Hexadezimaler Wert

Die Auswertung von Escape-Zeichen kann man verhindern, indem man einem String ein r oder R unmittelbar voranstellt.

Beispiel: r”\n bewirkt einen Zeilenvorschub”

Byte-Strings

Python 3 benutzt die Konzepte Text und (binäre) Daten, statt Strings und 8-bit Strings wie in Python 2.x. Jeder String oder Text in Python 3 ist Unicode, aber kodiertes Unicode wird als Datenstrings in Bytes dargestellt. Der Datentyp für Text ist str; der Datentyp für Daten ist bytes. Es ist nicht möglich Text und Daten (bytes) zu mischen. Versucht man zu mischen, erhält man den Ausnahmefehler TypeError. Während ein String-Objekt eine Folge von Zeichen in Unicode enthält, enthält ein Byte-Objekt eine Folge von Bytes mit den Werten 0 .. 255, die den ASCII-Werten entsprechen. Zur Verdeutlichung ein Beispiel:

```
x = "Hallo"
t = str(x)
u = t.encode("UTF-8")
print(u)
```

```
b'Hallo'
```

6 Python3 Operatoren

6.0.1 Ausdrücke und Operatoren

Ausdrücke

Ein Ausdruck kombiniert Literale, Namen, Funktionsaufrufe und Operatoren zu einem Wert. Beispiele sind `2 + 3`, `len(text) > 10` oder `name in teilnehmer`. Ein Ausdruck muss nicht zwingend einer Variablen zugewiesen werden.

Wichtige Operatoren

Operator	Bedeutung	Beispiel
`+`, `-`	Addition, Subtraktion	`10 - 3` → `7`
`*`	Multiplikation	`6 * 7` → `42`
`/`	True Division	`10 / 4` → `2.5`
`//`	Floor Division	`10 // 4` → `2`
`%`	Rest/Modulo	`27 % 7` → `6`
`**`	Potenzierung	`10 ** 3` → `1000`
`@`	Matrixmultiplikation für Typen, die sie implementieren	z. B. NumPy-Arrays
`==`, `!=`	Gleichheit/Ungleichheit	`2 == 2` → `True`
`<`, `<=`, `>`, `>=`	Größenvergleiche	`2 <= 3` → `True`
`is`, `is not`	Objektidentität	`x is None`
`in`, `not in`	Mitgliedschaft	`1 in [3, 2, 1]` → `True`
`and`, `or`, `not`	boolesche Verknüpfungen	`a and b`
`&`, `|`, `^`	bitweises Und, Oder, XOR	`6 ^ 3` → `5`
`~`	bitweise Invertierung	`~3` → `-4`
`<<`, `>>`	Bitverschiebung	`6 << 3` → `48`

/, // und %

Bei Python 3 liefert `/` bei der Division zweier Integerwerte einen `float`. Der Operator `//` berechnet dagegen die mathematische Floor Division: Das Ergebnis wird in Richtung minus unendlich abgerundet. Deshalb gilt:

```
10 / 3       # 3.3333333333333335
10 // 3      # 3
-10 // 3     # -4
int(-10 / 3) # -3
```

Floor Division und Modulo hängen zusammen. Für geeignete Zahlen `x` und `y != 0` gilt:

```
x == (x // y) * y + (x % y)
```

Gleichheit ist nicht Identität

`==` vergleicht Werte, `is` vergleicht Objektidentität. Für Zahlen, Strings, Listen oder andere normale Werte sollte man zur Wertprüfung `==` verwenden. `is` ist beispielsweise bei `x is None` die richtige Wahl.

Verkettete Vergleiche

Python erlaubt mathematisch gut lesbare Vergleichsketten:

```
0 <= x < 100
```

Das entspricht sinngemäß `0 <= x and x < 100`, wobei `x` in der Vergleichskette nur einmal ausgewertet wird.

Boolesche Operatoren

`not` liefert einen booleschen Wert. `and` und `or` liefern dagegen einen ihrer Operanden zurück. Dadurch sind Ausdrücke wie `name or Ünbekannt"` möglich. In Bedingungen entscheidet jeweils der Wahrheitswert (*truth value*) des Objekts.

7 Python3 Sequentielle Datentypen

7.0.1 Sequentielle Datentypen

Einführung

Sequenzen sind geordnete Sammlungen, auf deren Elemente typischerweise über Indizes zugegriffen werden kann. Zu den wichtigsten eingebauten Sequenztypen gehören `str`, `list`, `tuple`, `range`, `bytes` und `bytearray`. `memoryview` stellt zusätzlich eine Sicht auf binäre Daten bereit.

Die Typen unterscheiden sich insbesondere darin, ob sie veränderlich sind: Listen und Bytearrays sind **mutable**, Strings, Tupel, `range`-Objekte und `bytes` dagegen **immutable**.

Viele Operationen sind für verschiedene Sequenztypen gleich: `len()`, Indizierung, Slicing, Iteration und – soweit vom jeweiligen Typ unterstützt – Konkatenation und Wiederholung.

```
text = "Auf Listen und Strings kann über Indizes zugegriffen werden!"
print(text[0], text[11], text[-2])
```

```
A u n
```

Zugriff auf Listen:

```
lst = ["Wien", "London", "Paris", "Berlin", "Zürich", "Hamburg"]
print(lst[0])
print(lst[2])
print(lst[-1])
```

```
Wien
Paris
Hamburg
```

Im Gegensatz zu anderen Programmiersprachen verwendet Python die gleichen Syntax- und Funktionsnamen, um sequentielle Datentypen zu bearbeiten. Beispielsweise kann die Länge eines Strings, einer Liste und eines Tupels mit einer Funktion namens len () bestimmt werden:

```
Länder = ["Deutschland", "Schweiz", "Österreich",
              "Frankreich", "Belgien", "Niederlande",
              "England"]
len(Länder)  # die Länge der Liste; die Anzahl der Objekte
```

```
7
```

```
fib = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
len(fib)
```

```
10
```

Bytes

Ein `bytes`-Objekt ist eine **unveränderliche Sequenz von Ganzzahlen im Bereich 0 bis 255**. Seine Darstellung verwendet aus praktischen Gründen ASCII-Zeichen für druckbare Bytewerte, aber `bytes` ist kein Texttyp. Text wird in Python mit `str` dargestellt.

Zwischen Text und Bytes wird explizit konvertiert: `str.encode()` kodiert Text, `bytes.decode()` dekodiert Bytes. Für Web- und Datenaustausch ist UTF-8 sehr häufig die passende Kodierung.

```
s = "Glückliche Fügung"
s_bytes = s.encode('utf-8')
s_bytes
```

```
b'Gl\xc3\xbcckliche F\xc3\xbcgung'
```

Strings: Wertgleichheit und Identität

Strings (`str`) sind unveränderliche Sequenzen von Unicode-Codepoints. Zwei Strings vergleicht man inhaltlich mit `==` bzw. `!=`. Der Operator `is` prüft dagegen, ob zwei Namen **dasselbe Objekt** bezeichnen, und ist kein Ersatz für einen Stringvergleich. Implementierungsdetails wie String-Interning dürfen daher nicht zur Programmlogik benutzt werden.

Für Singletons wie `None` ist `is` dagegen die übliche Schreibweise, beispielsweise `x is None`.

```
text = "Python"
kopie = "".join(["Py", "thon"])
print(text == kopie)      # gleicher Wert
print(text is None)       # Identitätsvergleich mit einem Singleton
```

Python-Listen

Bisher haben wir bereits einige Listen verwendet, und hier folgt eine angemessene Einführung. Listen beziehen sich auf Arrays von Programmiersprachen wie C, C ++ oder Java, aber Python-Listen sind weitaus flexibler und leistungsfähiger als “klassische” Arrays. Beispielsweise müssen nicht alle Elemente in einer Liste denselben Typ haben. Darüber hinaus können Listen in einem Programmlauf wachsen, während in C die Größe eines Arrays zur Kompilierungszeit festgelegt werden muss.

Im Allgemeinen ist eine Liste eine Sammlung von Objekten. Genauer gesagt: Eine Liste in Python ist eine geordnete Gruppe von Elementen oder Elementen. Es ist wichtig zu beachten, dass diese Listenelemente nicht vom gleichen Typ sein müssen. Es kann eine beliebige Mischung von Elementen wie Zahlen, Zeichenfolgen, anderen Listen usw. sein. Der Listentyp ist für Python-Skripte und -Programme unerlässlich. Mit anderen Worten, Sie werden ohne Liste kaum ernsthaften Python-Code finden.

Die Haupteigenschaften von Python-Listen:

- `Sie sind ordentlich.`

- Sie enthalten beliebige Objekte.
- Auf Elemente einer Liste kann über einen Index zugegriffen werden.
- Sie können beliebig verschachtelt werden. Sie können andere Listen als Unterlisten enthal
- Variable Größe
- Sie sind veränderlich. Die Elemente einer Liste können geändert werden.

Listennotation und Beispiele

Listenobjekte werden in eckige Klammern eingeschlossen und durch Kommas getrennt. Die folgende Tabelle enthält einige Beispiele für Listen:

Listen

Beschreibung

[]

Eine leere Liste

[1, 1, 2, 3, 5, 8]

Eine Liste mit ganzen Zahlen

[42, “What’s the question?”, 3.1415]

Eine Liste mit gemischten Datentypen

[“Stuttgart”, “Freiburg”, “München”, “Nürnberg”, “Würzburg”, “Ulm”, “Friedrichshafen”, “Zürich”, “Wien”]

Eine Liste von Strings

[[“London”, “England”, 7556900], [“Paris”, “France”,2193031], [“Bern”, “Switzerland”, 123466]]

Eine verschachtelte Liste

[“Oberste Ebene”, [“Ein Level runter”, [“noch tiefer”, [“und tiefer”, 42]]]]

Eine tief verschachtelte Liste

Zugriff auf Listenelemente Zuweisen einer Liste zu einer Variablen:

Es gibt verschiedene Möglichkeiten, auf die Elemente einer Liste zuzugreifen. Der wahrscheinlich einfachste Weg für C-Programmierer sind Indizes. Die Nummern der Listen werden beginnend mit 0 aufgelistet:

```
Sprachen = ["Python", "C", "C++", "Java", "Perl"]
print(Sprachen[0] + " sind " + Sprachen[1] + " unterschiedlich!")
print("Zugriff auf das letzte Element der Liste: " + Sprachen[-1])
```

```
Python sind C unterschiedlich!
Zugriff auf das letzte Element der Liste: Perl
```

Das vorherige Beispiel einer Liste war eine Liste mit Elementen gleichen Datentyps. Wie wir bereits gesehen haben, können Listen verschiedene Datentypen haben, wie im folgenden Beispiel gezeigt:

Unterlisten

Listen können Unterlisten als Elemente enthalten. Diese Unterlisten können auch Unterlisten enthalten, d. H. Listen können durch Unterlistenstrukturen rekursiv erstellt werden.

```
Person = [["Marc", "Mayer"], ["17, Oxford Str", "12345", "London"],
↪  "07876-7876"]
Name = Person[0]
print(Name)
```

```
['Marc', 'Mayer']
```

```
Vor_name = Person[0][0]
print(Vor_Name)
```

```
Marc
```

```
Nach_name = Person[0][1]
print(Nach_name)
```

```
Mayer
```

```
Adresse = Person[1]
Straße = Person[1][0]
print(Straße)
```

```
17, Oxford Str
```

Das nächste Beispiel zeigt eine komplexere Liste mit einer tief strukturierten Liste:

```
komplexe_Liste = [["a", ["b", ["c", "x"]]]]
komplexe_Liste = [["a", ["b", ["c", "x"]]], 42]
komplexe_Liste[0][1]
```

```
['b', ['c', 'x']]
```

```
komplexe_Liste[0][1][1][0]
```

```
'c'
```

Listen ändern

```
Sprachen = ["Python", "C", "C++", "Java", "Perl"]
Sprachen[4] = "Lisp"
Sprachen
```

```
['Python', 'C', 'C++', 'Java', 'Lisp']
```

```
Sprachen.append("Haskell")
Sprachen
```

```
['Python', 'C', 'C++', 'Java', 'Lisp', 'Haskell']
```

```
Sprachen.insert(1, "Perl")
Sprachen
```

```
['Python', 'Perl', 'C', 'C++', 'Java', 'Lisp', 'Haskell']
```

```
Einkaufsliste = ['Milch', 'Joghurt', 'Ei', 'Butter', 'Brot', 'Bananen']
```

Wir gehen in einen virtuellen Supermarkt. Holen Sie sich einen Einkaufswagen und beginnen Sie mit dem Einkauf:

```
Einkaufsliste  = ['Milch', 'Joghurt', 'Ei', 'Butter', 'Brot', 'Bananen']
Einkaufswagen = []
#  "pop()"" Entfernt das letzte Element der Liste und gibt es zurück
Artikel = Einkaufsliste.pop()
print(Artikel, Einkaufsliste)
Einkaufswagen.append(Artikel)
print(Einkaufswagen)

# wir machen so weiter:
Artikel  = Einkaufsliste.pop()
print("shopping_list:", Einkaufsliste)
Einkaufswagen.append(Artikel)
print("cart: ", Einkaufswagen)
```

```
Bananen ['Milch', 'Joghurt', 'Ei', 'Butter', 'Brot']
['Bananen']
shopping_list: ['Milch', 'Joghurt', 'Ei', 'Butter']
cart:  ['Bananen', 'Brot']
```

Mit einer while-Schleife:

```
Einkaufsliste = ['milk', 'yoghurt', 'egg', 'butter', 'bread', 'bananas']
Einkaufswagen = []
while Einkaufsliste != []:
    Artikel = Einkaufsliste.pop()
    Einkaufswagen.append(Artikel)
    print(Artikel, Einkaufsliste, Einkaufswagen)

print("Einkaufsliste: ", Einkaufsliste)
print("Einkaufswagen: ", Einkaufswagen)
```

```
bananas ['milk', 'yoghurt', 'egg', 'butter', 'bread'] ['bananas']
bread ['milk', 'yoghurt', 'egg', 'butter'] ['bananas', 'bread']
butter ['milk', 'yoghurt', 'egg'] ['bananas', 'bread', 'butter']
egg ['milk', 'yoghurt'] ['bananas', 'bread', 'butter', 'egg']
yoghurt ['milk'] ['bananas', 'bread', 'butter', 'egg', 'yoghurt']
milk [] ['bananas', 'bread', 'butter', 'egg', 'yoghurt', 'milk']
Einkaufsliste:  []
Einkaufswagen:  ['bananas', 'bread', 'butter', 'egg', 'yoghurt', 'milk']
```

Tupel

Ein Tupel ist eine **unveränderliche Sequenz**. Es ähnelt in vielen Operationen einer Liste, ist aber nicht einfach eine „unveränderliche Liste“: Tupel werden häufig für feste Datensätze mit mehreren Feldern verwendet, etwa `(name, alter, ort)`.

Entscheidend für die Tupel-Syntax ist das **Komma**, nicht die Klammer. So ist `42,` bereits ein 1-Tupel; Klammern werden meist zur besseren Lesbarkeit verwendet.

Die Struktur eines Tupels kann nach seiner Erzeugung nicht verändert werden. Seine Elemente können jedoch selbst veränderliche Objekte sein. Deshalb ist auch nicht jedes Tupel hashbar. Ein Tupel kann nur dann beispielsweise als Dictionary-Schlüssel oder Set-Element verwendet werden, wenn alle für den Hash relevanten enthaltenen Objekte ebenfalls hashbar sind.

```
t = ("Tupels", "sind", "unveränderlich")
t[0]
```

```
'Tupels'
```

```
t[0] = "Zuordnungen zu Elementen sind nicht möglich"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-055c11c313f3> in <module>
----> 1 t[0] = "Zuordnungen zu Elementen sind nicht möglich"

TypeError: 'tuple' object does not support item assignment
```

Ausschneiden / Slicing

In vielen Programmiersprachen kann es ziemlich schwierig sein, einen Teil eines Strings zu schneiden, und noch schwieriger, wenn Sie ein “Subarray” ansprechen möchten. Python macht es mit seinem Slice-Operator sehr einfach. Slicing wird häufig in anderen Sprachen als Funktion mit möglichen Namen wie “Teilstring”, “gstr” oder “substr” implementiert.

Jedes Mal, wenn Sie einen Teil eines Strings oder einer Liste in Python extrahieren möchten, sollten Sie den Slice-Operator verwenden. Die Syntax ist einfach. Eigentlich sieht es ein bisschen so aus, als würde man auf ein einzelnes Element mit einem Index zugreifen, aber statt nur einer Zahl haben wir mehr, getrennt durch einen Doppelpunkt “:”. Wir haben einen Start- und einen Endindex, einer oder beide fehlen möglicherweise. Es ist am besten, die Funktionsweise von Slice anhand von Beispielen zu untersuchen:

```
Slogan = "Python ist großartig"
erste_fünf = Slogan[0:5]
erste_fünf
```

```
'Pytho'
```

```
ab_fünf = Slogan[5:]
ab_fünf
```

```
'n ist großartig'
```

```
eine_Kopie = Slogan[:]
ohne_letzte_fünf = Slogan[0:-5]
ohne_letzte_fünf
```

```
'Python ist groß'
```

Syntaktisch gibt es keinen Unterschied bei Listen. Wir werden mit europäischen Städtenamen zu unserem vorherigen Beispiel zurückkehren:

```
Städte = ["Wien", "London", "Paris", "Berlin", "Zürich", "Hamburg"]
erste_drei = Städte[0:3]
# oder einfacher:

erste_drei = Städte[:3]
print(erste_drei)
```

```
['Wien', 'London', 'Paris']
```

Jetzt extrahieren wir alle Städte außer den letzten beiden, "Zürich" und "Hamburg":

```
alles_außer_letzte_zwei = Städte[:-2]
print(alles_außer_letzte_zwei)
```

```
['Wien', 'London', 'Paris', 'Berlin']
```

Das Slicing kann ein drittes Argument, die Schrittweite, verwenden:

```
s[start:stop:step]
```

`start` ist inklusive, `stop` exklusiv. Eine positive Schrittweite läuft nach rechts; eine negative Schrittweite kann eine Sequenz rückwärts durchlaufen. Fehlende Grenzen werden passend zur Richtung ergänzt.

Für positive Schrittweite werden also die Indizes `start`, `start + step`, `start + 2*step`, … verwendet, solange der jeweilige Index vor `stop` liegt. Negative Indizes und negative Schrittweiten werden von Python entsprechend den allgemeinen Slice-Regeln normalisiert.

Im folgenden Beispiel definieren wir eine Zeichenfolge und drucken jedes dritte Zeichen dieser Zeichenfolge:

```
Slogan = "Python unter Linux ist großartig"
Slogan[::3]
```

```
'Ph t n trai'
```

Die folgende Zeichenfolge, die wie ein Buchstabensalat aussieht, enthält zwei Sätze. Eine davon enthält verdeckte Werbung für meine Python-Kurse in Kanada:

Versuchen Sie, den versteckten Satz mit Slicing herauszufinden. Die Lösung besteht aus 2 Schritten!

```
s =
↪  "TPoyrtohnotno  ciosu rtshees  lianr gTeosrto nCtiot yb yi nB oCdaennasdeao"
print(s)
```

```
TPoyrtohnotno  ciosu rtshees  lianr gTeosrto nCtiot yb yi nB oCdaennasdeao
```

```
s[::2]
```

```
'Toronto is the largest City in Canada'
```

```
s[1::2]
```

```
'Python courses in Toronto by Bodenseo'
```

Vielleicht interessiert Sie auch, wie wir den String erstellt haben. Sie müssen das Listenverständnis verstehen, um Folgendes zu verstehen:

```
s = "Toronto is the largest City in Canada"
t = "Python courses in Toronto by Bodenseo"
s = "".join(["".join(x) for x in zip(s,t)])
s
```

```
'TPoyrtohnotno  ciosu rtshees  lianr gTeosrto nCtiot yb yi nB oCdaennasdeao'
```

Länge

Die Länge einer Sequenz, d. H. Einer Liste, eines Strings oder eines Tupels, kann mit der Funktion len () bestimmt werden. Bei Zeichenfolgen wird die Anzahl der Zeichen und bei Listen oder Tupeln die Anzahl der Elemente gezählt, während eine Unterliste als ein Element zählt.

```
txt = "Hallo Welt"
len(txt)
```

```
10
```

```
a = ["Swen", 45, 3.54, "Basel"]
len(a)
```

```
4
```

Verkettung von Sequenzen

Das Kombinieren von zwei Sequenzen wie Zeichenfolgen oder Listen ist so einfach wie das Addieren von zwei Zahlen. Sogar das Bedienerzeichen ist das gleiche. Das folgende Beispiel zeigt, wie zwei Zeichenfolgen zu einer verkettet werden:

```
Vorname = "Homer"
```

```
Nachname = "Simpson"
Name = Vorname + " " + Nachname
print(Name)
```

```
Homer Simpson
```

So einfach ist das für Listen:

```
Farben1 = ["rot", "grün","blau"]
Farben2 = ["schwarz", "weiß"]
Farben = Farben1 + Farben2
print(Farben)
```

```
['rot', 'grün', 'blau', 'schwarz', 'weiß']
```

Die erweiterte Zuweisung `+=` ist **nicht allgemein dasselbe** wie `s = s + t`. Der linke Zielausdruck wird nur einmal ausgewertet, und veränderliche Typen können die Operation in-place ausführen.

Bei Listen erweitert `+=` die vorhandene Liste typischerweise um die Elemente des Iterables. Bei unveränderlichen Sequenzen wie Strings und Tupeln entsteht dagegen ein neues Objekt.

Überprüfen, ob ein Element enthalten ist

Mit `in` und `not in` lässt sich die Zugehörigkeit prüfen.

```
abc = ["a","b","c","d","e"]
"a" in abc
```

```
True
```

```
"a" not in abc
```

```
False
```

```
"e" not in abc
```

```
False
```

```
"f" not in abc
```

```
True
```

```
Slogan = "Python ist einfach!"
"y" in Slogan
```

```
True
```

```
"x" in Slogan
```

```
False
```

Wiederholungen

Bisher hatten wir einen "+" - Operator für Sequenzen. Es ist auch ein "∗" - Operator verfügbar. Natürlich ist keine "Multiplikation" zwischen zwei Sequenzen möglich. "*" ist für eine Sequenz und eine ganze Zahl definiert, d. h. "s ∗ n" oder "n ∗ s". Es ist eine Art Abkürzung für eine n-fache Verkettung, d.h.

str ∗ 4

ist das gleiche wie

str + str + str + str

Weitere Beispiele:

```
3 * "xyz-"
```

```
'xyz-xyz-xyz-'
```

```
"xyz-" * 3
```

```
'xyz-xyz-xyz-'
```

```
3 * ["a","b","c"]
```

```
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

Die erweiterte Zuordnung für "∗" kann ebenfalls verwendet werden: `s *= n` ist dasselbe wie `s = s * n`.

Die Fallstricke von Wiederholungen

In unseren vorherigen Beispielen haben wir den Wiederholungsoperator auf Zeichenfolgen und flache Listen angewendet. Wir können es auch auf verschachtelte Listen anwenden:

```
x = ["a","b","c"]
y = [x] * 4
y
```

```
[['a', 'b', 'c'], ['a', 'b', 'c'], ['a', 'b', 'c'], ['a', 'b', 'c']]
```

```
y[0][0] = "p"
y
```

```
[['p', 'b', 'c'], ['p', 'b', 'c'], ['p', 'b', 'c'], ['p', 'b', 'c']]
```

Dieses Ergebnis ist für Anfänger der Python-Programmierung ziemlich erstaunlich. Wir haben dem ersten Element der ersten Unterliste von y einen neuen Wert zugewiesen, dh y [0] [0], und wir haben die ersten Elemente aller Unterlisten in y "automatisch" geändert, dh y [1] [0]. , y [2] [0], y [3] [0].

Der Grund ist, dass der Wiederholungsoperator `* 4` 4 Verweise auf die Liste x erstellt: und

es ist daher klar, dass jedes Element von y geändert wird, wenn wir einen neuen Wert auf y [0] [0] anwenden.

8 Python3 Listen

8.0.1 Listen

Listen ändern

Python-Listen sind geordnete, veränderbare Sequenzen. In diesem Kapitel geht es vor allem darum, Elemente anzuhängen, einzufügen, zu suchen und zu löschen. Dabei ist wichtig zu unterscheiden, ob eine Operation die vorhandene Liste verändert oder eine neue Liste erzeugt.

Eine Liste kann man als einen Stapelspeicher (englisch: stack) ansehen. In der Informatik bezeichnet ein Stapelspeicher (oder Stapel) – manchmal auch Kellerspeicher (Keller) genannt – eine Datenstruktur mit minimal zwei Operationen: eine, mit der man Daten auf den Stapel legen kann, und eine, um das oberste Element des Stapels wegzunehmen. Stellen Sie sich das wie einen Stapel noch zu lesender Bücher vor. Sie wollen die Bücher des Stapels von oben nach unten durchlesen. Kaufen Sie zwischendurch ein neues Buch, kommt es oben drauf und ist damit das nächste Buch, was gelesen werden "muss". In den Programmiersprachen werden hierfür meistens die folgenden Operationen zur Verfügung gestellt:

- push Diese Methode dient dazu ein neues Objekt auf den Stapel zu legen. Je nach Sichtweise wird das Objekt "oben" oder "rechts" angefügt. Im Deutschen gibt es für push eine eher selten benutzte Übersetzung nämlich "einkellern". Python bietet im Gegensatz zu vielen anderen Sprachen keine Methode mit dem Namen "push", aber "append" erfüllt die gleiche Funktionalität.
- pop Diese Methode liefert das oberste Objekt eines Stapels zurück. Dabei wird das Objekt vom Stapel entfernt. Man bezeichnet dies im Deutschen als "auskellern".
- peek Diese Methode dient zum Nachschauen, was zuoberst auf dem Stapel liegt. Dabei wird das Objekt aber nicht wie bei pop entfernt. In Python gibt es keine solche Methode. Bei Listen oder Tupel kann man sie jedoch mit einem Indexzugriff simulieren. Falls "liste" eine Liste ist, dann verhält sich liste[-1] wie eine Methode liste.peek(), die es aber in der list-Klasse nicht gibt.

Stack und Queue: Mit `append()` und `pop()` eignet sich eine Liste gut als LIFO-Stack. Für eine FIFO-Queue, bei der häufig am Anfang eingefügt oder entfernt wird, ist `collections.deque` in der Regel die passendere Datenstruktur.

pop und append

- s.append(x) Hängt x ans Ende der Liste s an. Dies entspricht der Methode "push", so wie man sie in anderen Programmiersprachen wie zum Beispiel Perl findet.
- s.pop(i) Gibt das i-te Element von s zurück und entfernt es dabei aus der Liste. Normalerweise, also in anderen Sprachen, liefert pop nur das "oberste" bzw. das am weitesten rechts stehende Element zurück.
- s.pop() Ist i nicht angegeben, wird das letzte Element genommen, was dem üblichen pop anderer Programmiersprachen entspricht.

```
l = [42,98,77]
l.append(103)
l
```

```
[42, 98, 77, 103]
```

```
x = l.pop()
x
```

```
103
```

```
l.pop()
```

```
77
```

Man kommt schnell in die Situation, dass man mehr als ein Element an eine Liste anhängen will. So möchte man beispielsweise die Elemente einer Liste anhängen. Versucht man dies mit append, erlebt man eine unangenehme Überraschung:

```
l = [42,98,77]
l2 = [8,69]
l.append(l2)
l
```

```
[42, 98, 77, [8, 69]]
```

Eigentlich hatten wir dieses Ergebnis “erwartet”.

extend

Für diese Fälle gibt es die extend Methode für Listen. Sie dient dazu, an eine Liste mehrere Elemente anzuhängen:

```
l = [42,98,77]
l2 = [8,69]
l.extend(l2)
l
```

```
[42, 98, 77, 8, 69]
```

Das Argument von extend muss ein iterierbares Objekt sein. So kann man extend beispielsweise auch auf Tupel und Strings anwenden:

```
l = [42,98,77]
l.extend("Hallo")
l
```

```
[42, 98, 77, 'H', 'a', 'l', 'l', 'o']
```

```
l = [42,98,77]
```

```
l.extend((3,4,5))
l
```

```
[42, 98, 77, 3, 4, 5]
```

Listen verketten mit + und +=

Der Operator `+` verkettet zwei Listen und erzeugt dabei eine *neue* Liste. Das ist semantisch etwas anderes als `append()`, das genau ein Objekt an die vorhandene Liste anhängt. Mit `+=` wird eine Liste dagegen in-place erweitert, ähnlich wie mit `extend()`:

```
L = [3,4]
L = L + [42]
L
```

```
[3, 4, 42]
```

Was das Laufzeitverhalten betrifft, ist bei diesem Weg höchste Vorsicht geboten, wie wir im Folgenden sehen werden. Eine weitere Möglichkeit besteht in der Verwendung der erweiterten Zuweisung (englisch: augmented assignment, compound assignment):

```
L = [3,4]
L += [42]
L
```

```
[3, 4, 42]
```

Logisch gesehen sind beide Vorgehensweisen gleichwertig, d.h. sie liefern die gleichen Ergebnisse. Im Folgenden wollen wir uns das Laufzeitverhalten dieser beiden und der append-Methode im Vergleich anschauen. Wir messen die Laufzeit mittels des time-Modules, auf das wir hier nicht weiter eingehen wollen. Für das Verständnis des Programmes genügt es zu wissen, dass time.time() eine Floatzahl zurückliefert, die die Zeit in Sekunden seit „The Epoch'' 1 darstellt. Mit time.time() - start_time berechnen wir also die Zeit in Sekunden, die nötig war die for-Schleifen zu berechnen.

```
import time

n= 100000

start_time = time.time()
l = []
for i in range(n):
    l = l + [i * 2]
print(time.time() - start_time)
```

```
29.950933933258057
```

```
start_time = time.time()
l = []
for i in range(n):
    l += [i * 2]
print(time.time() - start_time)
```

```
0.05884385108947754
```

```
start_time = time.time()
l = []
for i in range(n):
    l.append(i * 2)
print(time.time() - start_time)
```

```
0.042885780334472656
```

Dieses Programm liefert “erschreckende” Ergebnisse.

Die konkreten Messwerte hängen von Python-Version, Rechner und Systemlast ab. Entscheidend ist das Prinzip: `l = l + [wert]` erzeugt in jedem Schleifendurchlauf eine neue Liste und kopiert die bisherigen Elemente. `append()` verändert die vorhandene Liste und ist für das schrittweise Hinzufügen einzelner Elemente die passende Operation. `+=` erweitert eine Liste in-place und vermeidet ebenfalls das wiederholte Erzeugen immer größerer Zwischenlisten.

Entfernen eines Wertes mit remove

Mit der Methode “remove” kann man einen bestimmten Wert ohne Kenntnis des Indexes aus einer Liste entfernen.

```
s.remove(x)
```

Dieser Aufruf entfernt das erste Vorkommen des Wertes x aus der Liste s. Falls x beim Aufruf nicht in der Liste vorhanden ist, gibt es einen ValueError. Im folgenden Beispiel rufen wir dreimal die remove-Methode auf, um die Farbe “green” zu entfernen. Da die Farbe nur zweimal in der Liste “colours” ist, erhalten wir beim dritten Mal einen ValueError:

```
Farben = ["rot", "grün", "blau", "grün", "gelb"]
Farben.remove("grün")
Farben
```

```
['rot', 'blau', 'grün', 'gelb']
```

```
Farben.remove("grün")
Farben
```

```
['rot', 'blau', 'gelb']
```

```
Farben.remove("green")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-12-7d960620585b> in <module>
----> 1 Farben.remove("green")

ValueError: list.remove(x): x not in list
```

Finden der Position eines Elementes

Mit der Methode index kann man die Position eines Elements innerhalb einer Liste ermitteln.

Es wird der Index für das x ermittelt. Falls der optionale Parameter i gegeben ist, beginnt die Suche erst ab dieser Position und endet bei der Position j, falls j gegeben ist. Es erfolgt eine Fehlermeldung, falls x nicht in s vorkommt.

```
Farben = ["red", "green", "blue", "green", "yellow"]
Farben.index("green")
```

```
1
```

```
Farben.index("green", 2)
```

```
3
```

```
Farben.index("green", 3,4)
```

```
3
```

```
Farben.index("black")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-4-b8696f57e410> in <module>
----> 1 Farben.index("black")

ValueError: 'black' is not in list
```

insert

Wir haben gesehen, dass wir mit append ein Element an das Ende einer Liste anhängen können. Aber es gibt keine Möglichkeit, mittels append ein Element an eine beliebige Stelle einer Liste einzufügen. Dazu gibt es die Methode “insert”:

Ein Objekt “object” wird in die Liste “s” eingefügt. Die früheren Elemente ab der Position index werden um eins nach rechts verschoben.

```
Farben = ["rot", "grün", "blau", "gelb"]
Farben.insert(1,"schwarz")
Farben
```

```
['rot', 'schwarz', 'grün', 'blau', 'gelb']
```

Das Verhalten der Methode “append” lässt sich sehr einfach mit “insert” simulieren:

```
abc = ["a","b","c"]
abc.insert(len(abc),"d")
abc
```

```
['a', 'b', 'c', 'd']
```

Anmerkungen:

1 “The Epoch” bezeichnet das Datum 1. Januar 1970 und die Uhrzeit 00:00, als Beginn der Unix Zeitzählung (ab UNIX Version 6)

9 Python3 Sets Mengen

9.0.1 Mengen

Einführung

Auch wenn die Mengenlehre über lange Zeit heftig kritisiert wurde und immer noch kritisiert wird, ist sie ein wesentliches Gebiet der Mathematik. Die heutige Mathematik ist in der Terminologie der Mengenlehre formuliert und baut auf deren Axiomen auf. Die Mengenlehre ist noch ein recht junges Gebiet der Mathematik. Der deutsche Mathematiker Georg Cantor (1845 - 1918) begründete die Mengenlehre mit seinem 1874 erschienenen Artikel "Über eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen". Anfangs also bis ins Jahr 1877 bezeichnete er übrigens noch die Mengenlehre als "Mannigfaltigkeitslehre".

1895 gab er folgende Definition einer Menge: "Unter einer 'Menge' verstehen wir jede Zusammenfassung M von bestimmten wohlunterschiedenen Objekten m unserer Anschauung oder unseres Denkens (welche die 'Elemente' von M genannt werden) zu einem Ganzen."

Eine Menge kann beliebige Elemente enthalten: zum Beispiel Zahlen, Zeichen, Buchstaben, Wörter, Namen oder sogar andere Mengen. Eine Menge wird in der Mathematik üblicherweise mit einem Großbuchstaben bezeichnet.

Mengen in Python

Der Datentyp "set", der ein sogenannter "collection"-Typ ist, ist in Python seit Version 2.4. enthalten. Ein set enthält eine ungeordnete Sammlung von einmaligen und unveränderlichen Elementen. In anderen Worten: Ein Element kann in einem set-Objekt nicht mehrmals vorkommen, was bei Listen und Tupel jedoch möglich ist. Beim Datentyp set handelt es sich um die Python-Implementierung von Mengen, wie sie aus der Mathematik bekannt sind.

Sets erzeugen

Will man eine Menge erzeugen, so ist dies in Python3 sehr einfach. Man bedient sich der gewohnten mathematischen Schreibweise.

```
staedte = {'Hamburg', 'München', 'Frankfurt', 'Berlin'}
print(staedte)
```

```
{'Hamburg', 'Frankfurt', 'München', 'Berlin'}
```

```
'Berlin' in staedte
```

```
True
```

```
'Köln' in staedte
```

```
False
```

Wenn man sich das obige Beispiel genau anschaut, fällt auf, dass `print` die Elemente nicht notwendigerweise in der Reihenfolge der Definition ausgibt. Python-Sets sind **ungeordnete Sammlungen**: Sie speichern keine Elementpositionen und garantieren keine Einfügereihenfolge. Auf eine beobachtete Iterations- oder Ausgabereihenfolge sollte man sich deshalb nicht verlassen.

Set-Elemente müssen **hashbar** sein. Listen und Dictionaries sind beispielsweise nicht hashbar und können deshalb keine Set-Elemente sein. „Unveränderlich“ ist als Faustregel oft hilfreich, aber technisch ist Hashbarkeit die entscheidende Eigenschaft.

```
staedte = { ['Berlin', 3.42], ['Hamburg', 1.75],  ['München', 1.41], ['Köln',
↪  1.03], ['Frankfurt', 0.70], ['Stuttgart', 0.60]}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-67-a4cdbbefd7dd> in <module>
----> 1 staedte = { ['Berlin', 3.42], ['Hamburg', 1.75],  ['München', 1.41],
↪  ['Köln', 1.03], ['Frankfurt', 0.70], ['Stuttgart', 0.60]}

TypeError: unhashable type: 'list'
```

Mit Tupeln funktioniert es nur dann, wenn das jeweilige Tupel selbst hashbar ist. Ein Tupel, das beispielsweise eine Liste enthält, ist ebenfalls nicht hashbar.

```
staedte = {('Frankfurt', 0.7), ('Stuttgart', 0.6), ('Hamburg', 1.75),
↪  ('Berlin', 3.42), ('Köln', 1.03), ('München', 1.41)}
```

Neben Set-Literalen mit geschweiften Klammern gibt es den Konstruktor `set()`. Er akzeptiert ein **beliebiges Iterable**, nicht nur Sequenzen. So kann man etwa eine Liste oder einen String in eine Menge eindeutiger Elemente überführen.

Eine leere Menge schreibt man als `set()`. Die Schreibweise `{}` erzeugt dagegen ein leeres Dictionary.

```
liste_von_woertern = ["gut", "hilfreich", "besser", "optimal"]
menge_von_woertern = set(liste_von_woertern)
menge_von_woertern
```

```
{'besser', 'gut', 'hilfreich', 'optimal'}
```

Häufig wird die set-Funktion angewendet, um aus Listen oder anderen sequentiellen Datentypen mehrfache Vorkommen zu entfernen. Wir wollen dies in einem Beispiel mit einer Liste von Städten aus der Schweiz, Österreich und Deutschland zeigen. In dieser Liste erscheint die Stadt Zürich zweimal. In der erzeugten Menge erscheint sie wie erwartet nur noch einmal, da es in einer Menge keine mehrfachen Vorkommen geben kann:

```
liste_von_staedten = ["Frankfurt", "Zürich", "Bern", "Stuttgart", "Freiburg",
↪  "Ulm", "Hamburg", "München", "Nürnberg", "Zürich", "Bregenz", "Salzburg",
↪  "Wien"]
```

```
liste_von_staedten
```

```
['Frankfurt',
 'Zürich',
 'Bern',
 'Stuttgart',
 'Freiburg',
 'Ulm',
 'Hamburg',
 'München',
 'Nürnberg',
 'Zürich',
 'Bregenz',
 'Salzburg',
 'Wien']
```

```
menge_von_staedten = set(liste_von_staedten)
menge_von_staedten
```

```
{'Bern',
 'Bregenz',
 'Frankfurt',
 'Freiburg',
 'Hamburg',
 'München',
 'Nürnberg',
 'Salzburg',
 'Stuttgart',
 'Ulm',
 'Wien',
 'Zürich'}
```

Anschließend kann man dann wieder die Menge in eine Liste wandeln, falls man eine Liste als Datentyp braucht:

```
liste_von_staedten = list(menge_von_staedten)
liste_von_staedten
```

```
['Hamburg',
 'Frankfurt',
 'München',
 'Wien',
 'Ulm',
 'Salzburg',
 'Bern',
 'Stuttgart',
 'Bregenz',
 'Freiburg',
 'Nürnberg',
 'Zürich']
```

Wie wir sehen können, hat diese Vorgehensweise einen kleinen Schönheitsfehler. Mehrfache Vorkommen sind zwar nun nicht mehr in der Ergebnisliste enthalten, aber die ursprüngliche Reihenfolge ist in der Regel nicht mehr vorhanden.

Wir kommen nun zu einem weiteren Anwendungsfall der set-Funktion. Im folgenden Beispiel

benutzen wir sie, um einen String in seine Zeichen zu vereinzeln:

```
x = set("Ein gutes Python-Tutorial")
x
```

```
{' ',
 '-',
 'E',
 'P',
 'T',
 'a',
 'e',
 'g',
 'h',
 'i',
 'l',
 'n',
 'o',
 'r',
 's',
 't',
 'u',
 'y'}
```

```
type(x)
```

```
set
```

Mengen aus hashbaren Elementen

Die Elemente eines Sets müssen hashbar sein. Deshalb sind Listen als Elemente nicht erlaubt. Hashbare Tupel, Strings, Zahlen und `frozenset`-Objekte sind typische Beispiele zulässiger Set-Elemente.

```
Städte = set((("Python","Perl"), ("Paris", "Berlin", "London")))
Städte = set((["Python","Perl"], ["Paris", "Berlin", "London"]))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-59-0e894acf2c44> in <module>
      1 Städte = set((("Python","Perl"), ("Paris", "Berlin", "London")))
----> 2 Städte = set((["Python","Perl"], ["Paris", "Berlin", "London"]))

TypeError: unhashable type: 'list'
```

Frozensets

Ein `set` ist selbst veränderlich und deshalb nicht hashbar. Ein `frozenset` enthält ebenfalls eindeutige hashbare Elemente, kann nach seiner Erzeugung aber nicht mehr verändert werden. Dadurch ist ein `frozenset` selbst hashbar und kann beispielsweise als Dictionary-Schlüssel oder als Element eines anderen Sets dienen.

```
Städte = {"Frankfurt", "Basel","Freiburg"}
```

```
Städte.add("Straßburg")
Städte
```

Frozensets besitzen die nicht-verändernden Mengenoperationen wie Vereinigung, Schnitt oder Differenz, aber keine mutierenden Methoden wie `add()` oder `remove()`:

```
Städte = frozenset(["Frankfurt", "Basel","Freiburg"])
Städte.add("Straßburg")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-57-f55e3ae660bc> in <module>
      1 Städte = frozenset(["Frankfurt", "Basel","Freiburg"])
----> 2 Städte.add("Straßburg")

AttributeError: 'frozenset' object has no attribute 'add'
```

Operationen auf `set`-Objekten

`add`-Methode Mit `add()` fügt man ein **hashbares** Objekt in ein Set ein, falls es noch nicht vorhanden ist. Ein erneutes Hinzufügen desselben Wertes verändert das Set nicht.

```
Farben = {"rot","grün"}
Farben.add("gelb")
Farben
```

```
{'gelb', 'grün', 'rot'}
```

```
Farben.add(["schwarz","weiß"])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-56-7098b094f288> in <module>
----> 1 Farben.add(["schwarz","weiß"])

TypeError: unhashable type: 'list'
```

Selbstverständlich wird ein Objekt nur dann als neues Element eingefügt, wenn es noch nicht enthalten ist. Ist es bereits enthalten, hat der Aufruf der Methode keine Auswirkungen.

clear-Methode Alle Elemente einer Menge werden entfernt. Die Menge ist also anschließend leer, wie wir im folgenden Beispiel sehen:

```
cities = {"Stuttgart", "Konstanz", "Freiburg"}
cities.clear()
cities
```

```
set()
```

copy-Methode copy erzeugt eine flache Kopie einer Menge, die zurückgeliefert wird. Wir demonstrieren die Benutzung anhand eines Beispiels:

```
more_cities = {"Winterthur","Schaffhausen","St. Gallen"}
cities_backup = more_cities.copy()
more_cities.clear()
cities_backup
```

```
{'Schaffhausen', 'St. Gallen', 'Winterthur'}
```

Nur für diejenigen, die glauben, dass eine einfache Zuweisung auch genügen könnte:

```
more_cities = {"Winterthur", "Schaffhausen", "St. Gallen"}
cities_backup = more_cities
more_cities.clear()
cities_backup
```

```
set()
```

Die Zuweisung "cities_backup = more_cities" erzeugt nur einen Pointer, einen weiteren Namen für das gleiche Objekt.

difference-Methode Diese Methode liefert die Differenz von zwei oder mehr Mengen zurück. Wir illustrieren dies wie immer an einem Beispiel:

```
x = {"a","b","c","d","e"}
y = {"b","c"}
z = {"c","d"}
x.difference(y)
```

```
{'a', 'd', 'e'}
```

```
x.difference(y).difference(z)
```

```
{'a', 'e'}
```

Statt die Methode "difference" zu benutzen, hätten wir auch den Operator "-" benutzen können:

```
x - y
```

```
{'a', 'd', 'e'}
```

```
x - y - z
```

```
{'a', 'e'}
```

difference_update-Methode Die Methode "difference_update" entfernt alle Elemente einer anderen Menge aus einer Menge. "x.difference_update(y)" ist gleichbedeutend mit "x = x - y".

```
x = {"a","b","c","d","e"}
```

```
y = {"b","c"}
x.difference_update(y)
x
```

```
{'a', 'd', 'e'}
```

```
x = {"a","b","c","d","e"}
y = {"b","c"}
x = x - y
x
```

```
{'a', 'd', 'e'}
```

discard-Methode Beim Aufruf von discard(el) wird das Element el aus einer Menge entfernt, falls es enthalten ist. Falls el nicht in der Menge enthalten ist, passiert nichts.

```
x = {"a","b","c","d","e"}
x.discard("a")
x
```

```
{'b', 'c', 'd', 'e'}
```

```
x.discard("z")
x
```

```
{'b', 'c', 'd', 'e'}
```

remove-Methode Die Methode "remove" funktioniert wie discard(), aber falls el nicht in der Menge enthalten ist, wird ein Fehler generiert, d.h. ein KeyError:

```
x = {"a","b","c","d","e"}
x.remove("a")
x
```

```
{'b', 'c', 'd', 'e'}
```

```
x.remove("z")
x
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-42-c666360df5af> in <module>
----> 1 x.remove("z")
      2 x

KeyError: 'z'
```

union-Methode Die Methode "union" liefert die Vereinigung von zwei Mengen als eine neue Menge zurück, d.h. alle Elemente, die in beiden Mengen vorkommen.

```
x = {"a","b","c","d","e"}
y = {"c","d","e","f","g"}
x.union(y)
```

```
{'a', 'b', 'c', 'd', 'e', 'f', 'g'}
```

Um die Vereinigen zu bilden kann man auch das Pipe-Zeichen "|" als Operator verwenden:

```
x = {"a","b","c","d","e"}
y = {"c","d","e","f","g"}
x | y
```

```
{'a', 'b', 'c', 'd', 'e', 'f', 'g'}
```

intersection-Methode Mit der Methode intersection kann man die Schnittmenge von zwei Mengen bilden, wie wir im folgenden Beispiel sehen.

```
x = {"a","b","c","d","e"}
y = {"c","d","e","f","g"}
x.intersection(y)
```

```
{'c', 'd', 'e'}
```

Dies kann auch mit dem "&"-Zeichen formuliert werden:

```
x = {"a","b","c","d","e"}
y = {"c","d","e","f","g"}
x & y
```

```
{'c', 'd', 'e'}
```

isdisjoint-Methode Diese Methode liefert True zurück, wenn zwei Mengen eine leere Schnittmenge haben.

```
x = {"a","b","c"}
y = {"c","d","e"}
x.isdisjoint(y)
```

```
False
```

```
x = {"a","b","c"}
y = {"d","e","f"}
x.isdisjoint(y)
```

```
True
```

issubset-Methode x.issubset(y) liefert True zurück, falls x eine Untermenge von y ist. "<=" kann statt dem Aufruf der Methode verwendet werden. "<" prüft, ob es sich um eine

echte Untermenge handelt: Wenn x < y gilt, dann enthält y mindestens ein Element, das nicht in x enthalten ist.

```
x = {"a","b","c","d","e"}
y = {"c","d"}
x.issubset(y)
```

```
False
```

```
y.issubset(x)
```

```
True
```

```
x < y
```

```
False
```

```
y < x # y ist eine echte Untermenge von x
```

```
True
```

```
x < x # eine Menge kann nie eine echte Untermenge ihrer selbst sein.
```

```
False
```

```
x <= x
```

```
True
```

issuperset-Methode x.issuperset(y) liefert True zurück, falls x eine Obermenge von y ist. “>=” kann statt dem Aufruf der Methode verwendet werden. “>” prüft, ob es sich um eine echte Obermenge handelt: Wenn x > y gilt, dann enthält x mindestens ein Element, dass nicht in y enthalten ist.

```
x = {"a","b","c","d","e"}
y = {"c","d"}
x.issuperset(y)
```

```
True
```

```
x > y
```

```
True
```

```
x >= y
```

```
True
```

```
x >= x
```

```
True
```

```
x > x
```

```
False
```

```
x.issuperset(x)
```

```
True
```

pop pop() liefert ein beliebiges Element der Menge zurück. Dieses Element wird dabei aus der Menge entfernt. Die Methode erzeugt einen KeyError, falls die Menge leer ist.

```
x = {"a","b","c","d","e"}
```

```
x.pop()
```

```
'd'
```

```
x.pop()
```

```
'e'
```

```
x.pop()
```

```
'a'
```

```
x.pop()
```

```
'b'
```

```
x.pop()
```

```
'c'
```

Auch wenn "pop" die Elemente in der Reihenfolge der internen Repräsentierung der Menge zurückliefert, so stimmt dennoch die Formulierung "pop() liefert ein beliebiges Element der Menge zurück." Die Menge die wir im "Kopf" haben, lautet ja im obigen Beispiel {"a","b","c","d","e"} und die Reihenfolge in der pop() die Elemente zurückliefert ist 'd', 'e', 'a', 'b', 'c'. Dass diese Reihenfolge für diese Menge nicht immer gleich ist, kann man in der folgenden neuen interaktiven Python-Sitzung sehen:

```
x = {"a", "b", "c", "d", "e", "f"}
x.pop()
```

```
'f'
```

```
x.pop()
```

```
'd'
```

```
x.pop()
```

```
'e'
```

```
x.pop()
```

```
'a'
```

```
x.pop()
```

```
'b'
```

Keine definierte Reihenfolge

Sets haben keine definierte Elementreihenfolge. Die konkrete Reihenfolge bei Iteration oder Darstellung ergibt sich aus Implementierungsdetails der Hash-Tabelle und kann sich beispielsweise durch einen anderen Programmlauf, eine andere Python-Version oder Änderungen an der Menge unterscheiden. Sie ist **nicht zufällig im mathematischen Sinn**, aber sie ist auch keine zugesicherte Programmschnittstelle.

Wenn eine stabile Reihenfolge benötigt wird, muss sie explizit hergestellt werden, beispielsweise mit `sorted(menge)` für gegenseitig vergleichbare Elemente.

```
x = {"a","b","c","d","e"}
x
```

```
{'a', 'b', 'c', 'd', 'e'}
```

```
staedte = {'Hamburg', 'München', 'Frankfurt', 'Berlin'}
x = {"a","b","c","d","e"}
x
```

```
{'a', 'b', 'c', 'd', 'e'}
```

```
staedte = ['Hamburg', 'München', 'Frankfurt', 'Berlin']
x = {"a","b","c","d","e"}
x
```

```
{'a', 'b', 'c', 'd', 'e'}
```

10 Mengen Beispiel

10.0.1 Python-Mengen: Umfangreiches Beispiel

Einfache Computerlinguistik mit Python

Dieses Kapitel beschäftigt sich mit natürlicher Sprache und Literatur. Es ist auch ein Beispiel von einfacher statistischer Computerlinguistik. Aber in erster Linie geht es darum ein umfangreiches Beispiel und einen interessanten Anwendungsfall für Python-Mengen bieten. Neulinge in Python denken oft, dass Mengen nur ein Spielzeug für Mathematiker sind und dass es keinen wirklichen Anwendungsfall in der Programmierung gibt. Das Gegenteil ist der Fall. Es gibt mehrere Anwendungsfälle für die Mengen also den Datentyp 'set' in der Python-Programmierung. Sie werden z. B. verwendet, um Doubletten - mehrfach vorkommende Elemente - in einer Liste zu beseitigen, d. h. um eine Liste eindeutig zu machen.

Im folgenden Beispiel werden wir den Datentyp 'set' verwenden, um die verschiedenen Wörter zu bestimmen, die in einem Roman vorkommen. Unser Anwendungsfall dreht sich um einen Roman, der von vielen gelobt und von einigen enthusiastisch als der beste Roman in englischer Sprache des 20. Jahrhunderts betrachtet wird, aber auch als der am schwersten zu lesende angesehen wird. Wir sprechen über den Roman "Ulysses" von James Joyce. Wir werden nicht über die Schönheit der Sprache oder den Sprachstil sprechen. Wir werden den Roman untersuchen, indem wir uns die im Roman verwendeten Wörter genau ansehen. Unser Ansatz wird rein statistisch sein. Die Behauptung ist, dass James Joyce in seinem Roman mehr Wörter verwendet als jeder andere Autor. Tatsächlich ist sein Wortschatz größer als der aller anderen Autoren, vielleicht sogar als der von Shakespeare.

Außer dem Roman "Ulysses" werden wir noch die folgenden Romane verwenden "Sons and Lovers" von D.H. Lawrence, "The Way of All Flesh" von Samuel Butler, "Robinson Crusoe" von Daniel Defoe, "To the Lighthouse" von Virginia Woolf, "Moby Dick" von Herman Melville and the Short Story "Metamorphosis" von Franz Kafka.

Wörter in einem Text

Um alle Wörter aus dem Roman "Ulysses" herauszuschneiden, können wir die Funktion findall aus dem Modul "re" verwenden:

```
import re

# uns ist die Groß- und Kleinschreibung egal und wir verwenden Kleinbuchstaben:
ulysses_txt = open("books/james_joyce_ulysses.txt").read().lower()

words = re.findall(r"\b[\w-]+\b", ulysses_txt)
print(f"Der Roman Ulysses enthält {len(words)} Wörter.")
```

```
Der Roman Ulysses enthält 272452 Wörter.
```

Diese Zahl ergibt sich aus der Summe aller Wörter sowie der vielen Wörter, die mehrfach vorkommen:

```
for word in ["the", "while", "good", "bad", "ireland", "irish"]:
    outstr = f"Das Wort '{word}' kommt {words.count(word)}"
    outstr += f" Mal in dem Roman vor!"
    print(outstr)
```

```
Das Wort 'the' kommt 15112 Male in dem Roman vor!
Das Wort 'while' kommt 123 Male in dem Roman vor!
Das Wort 'good' kommt 321 Male in dem Roman vor!
Das Wort 'bad' kommt 90 Male in dem Roman vor!
Das Wort 'ireland' kommt 90 Male in dem Roman vor!
Das Wort 'irish' kommt 117 Male in dem Roman vor!
```

272452 Wörter sind sicherlich eine enorme Anzahl von Wörtern für einen Roman, aber auf der anderen Seite gibt es viele Romane mit noch mehr Wörtern. Interessanter und aussagekräftiger über die Qualität eines Romans ist die Anzahl der verschiedenen Wörter. Das ist der Moment, in dem wir endlich "Set" brauchen. Wir werden die Wortliste "words" in eine Menge verwandeln. Wenden wir "len" auf diese Menge an, erhalten wir die Anzahl der verschiedenen Wörter:

```
diff_words = set(words)
print(f"'Ulysses' enthält {len(diff_words)} verschiedene Wörter!")
```

```
'Ulysses' enthält 29422 verschiedene Wörter!
```

Das ist in der Tat eine beeindruckende Zahl. Das sieht man umso besser, wenn man es mit den Worthäufigkeiten der anderen unten stehenden Romane vergleicht:

```
novels = ['sons_and_lovers_lawrence.txt',
          'metamorphosis_kafka.txt',
          'the_way_of_all_flash_butler.txt',
          'robinson_crusoe_defoe.txt',
          'to_the_lighthouse_woolf.txt',
          'james_joyce_ulysses.txt',
          'moby_dick_melville.txt']

print(f"{'Name des Romans':38s}: Anz. verschiedene Wörter")
for novel in novels:
    txt = open("books/" + novel).read().lower()
    words = re.findall(r"\b[\w-]+\b", txt)
    diff_words = set(words)
    n = len(diff_words)
    print(f"{novel[:-4]:38s}: {n:5d}")
```

```
Name des Romans                       : Anz. verschiedene Wörter
sons_and_lovers_lawrence              : 10822
metamorphosis_kafka                   :  3027
the_way_of_all_flash_butler           : 11434
robinson_crusoe_defoe                 :  6595
to_the_lighthouse_woolf               : 11415
james_joyce_ulysses                   : 29422
moby_dick_melville                    : 18922
```

Besondere Wörter in Ulysses

In dem folgenden kleinen Python-Programm werden wir alle Wörter, die in den anderen Romanen vorkommen, von "Ulysses" subtrahieren. Es ist erstaunlich, wie viele Wörter von James Joyce verwendet werden, die keiner der anderen Autoren verwendet:

```
words_in_novel = {}
for novel in novels:
    txt = open("books/" + novel).read().lower()
    words = re.findall(r"\b[\w-]+\b", txt)
    words_in_novel[novel] = words

words_only_in_ulysses =  set(words_in_novel['james_joyce_ulysses.txt'])
novels.remove('james_joyce_ulysses.txt')
for novel in novels:
    words_only_in_ulysses -= set(words_in_novel[novel])

with open("books/words_only_in_ulysses.txt", "w") as fh:
    txt = " ".join(words_only_in_ulysses)
    fh.write(txt)

print(len(words_only_in_ulysses))
```

```
15314
```

Übrigens hat amerikanische Kinderbuchautor Dr. Seuss ein Buch mit nur 50 verschiedenen Wörtern geschrieben: "Green Eggs and Ham"

Die Datei mit den Wörtern, die nur in Ulysses vorkommen enthält seltsame oder selten verwendete Wörter wie:

huntingcrop tramtrack pappin kithogue pennyweight undergarments scission nagyaságos wheedling begad dogwhip hawthornden turnbull calumet covey repudiated pendennis waistcoatpocket nostrum

Gemeinsame Wörter

Es ist auch möglich, die Wörter zu finden, die in jedem Buch vorkommen. Dazu benötigen wir die Schnittmenge:

```
# Wir starten mit den Wörtern in ulysses
common_words = set(words_in_novel['james_joyce_ulysses.txt'])
for novel in novels:
    common_words &= set(words_in_novel[novel])

print(len(common_words))
```

```
1745
```

Richtig machen

Bei den vorherigen Berechnungen ist uns ein kleiner Fehler unterlaufen. Wenn Sie sich die Texte ansehen, werden Sie feststellen, dass sie einen Kopf- und Fußteil haben, der von

Project Gutenberg hinzugefügt wurde und nicht zu den Texten gehört. Die Texte sind zwischen den Zeilen positioniert:

START OF THE PROJECT GUTENBERG EBOOK THE WAY OF ALL FLESH

und

END OF THE PROJECT GUTENBERG EBOOK THE WAY OF ALL FLESH

oder

*** START OF THIS PROJECT GUTENBERG EBOOK ULYSSES ***

und

*** END OF THIS PROJECT GUTENBERG EBOOK ULYSSES ***

Die Funktion read_text nimmt sich dieses Problemes an, d.h. sie schneidet die Textblöcke aus:

```
def read_text(fname):
    beg_e = re.compile(r
    ↪  "\*\*\* ?start of (this|the) project gutenberg ebook[^*]*\*\*\*")
    end_e = re.compile(r
    ↪  "\*\*\* ?end of (this|the) project gutenberg ebook[^*]*\*\*\*")
    txt = open("books/" + fname).read().lower()
    beg = beg_e.search(txt).end()
    end = end_e.search(txt).start()
    return txt[beg:end]

words_in_novel = {}
for novel in novels + ['james_joyce_ulysses.txt']:
    txt = read_text(novel)
    words = re.findall(r"\b[\w-]+\b", txt)
    words_in_novel[novel] = words

words_in_ulysses =  set(words_in_novel['james_joyce_ulysses.txt'])
for novel in novels:
    words_in_ulysses -= set(words_in_novel[novel])

with open("books/words_in_ulysses.txt", "w") as fh:
    txt = " ".join(words_in_ulysses)
    fh.write(txt)

print(len(words_in_ulysses))


# we start with the words in ulysses
common_words = set(words_in_novel['james_joyce_ulysses.txt'])
for novel in novels:
    common_words &= set(words_in_novel[novel])

print(len(common_words))
```

```
15341
1279
```

Die Wörter der Menge “common_words” sind Wörter, die zu den am häufigsten verwendeten

Wörtern der englischen Sprache gehören. Schauen wir uns 30 beliebige Wörter aus dieser Menge an:

```
counter = 0
for word in common_words:
    print(word, end=", ")
    counter += 1
    if counter == 30:
        break
```

```
beginning, thanks, sense, sick, ceiling, around, out, directly, all, everything,
↪  breathing, pity, startled, strict, morning, quite, rather, line, turning,
↪  direct, losing, think, instead, lay, various, begin, handle, for, set,
↪  possible,
```

11 Python3 Deep Copy

11.0.1 Flache und tiefe Kopie

Einführung

In diesem Kapitel werden wir uns mit der Frage befassen, wie Listen und verschachtelte Listen kopiert werden. Die Probleme, auf die wir stoßen werden, sind allgemeine Probleme veränderlicher Datentypen. Der Versuch, Listen zu kopieren, kann für Neulinge eine verblüffende Erfahrung sein. Zuvor möchten wir jedoch einige Erkenntnisse aus dem vorherigen Kapitel Datentypen und Variablen zusammenfassen. Python zeigt sogar ein seltsames Verhalten für die Anfänger der Sprache - im Vergleich zu einigen anderen traditionellen Programmiersprachen - beim Zuweisen und Kopieren einfacher Datentypen wie Ganzzahlen und Zeichenfolgen. Der Unterschied zwischen flachem und tiefem Kopieren ist nur für zusammengesetzte Objekte relevant, d. H. Objekte, die andere Objekte enthalten, wie Listen oder Klasseninstanzen.

Im folgenden Codeausschnitt zeigt y auf denselben Speicherort wie X. Wir können dies sehen, indem wir die Funktion id () auf x und y anwenden. Im Gegensatz zu “echten” Zeigern wie in C und C ++ ändern sich die Dinge jedoch, wenn wir y einen neuen Wert zuweisen. In diesem Fall erhält y einen separaten Speicherort, wie wir im Kapitel Datentypen und Variablen gesehen haben und im folgenden Beispiel sehen können ::

```
x = 3
y = x
print(id(x), id(y))
```

```
140720897565136 140720897565136
```

```
y = 4
print(id(x), id(y))
```

```
140720897565136 140720897565168
```

```
print(x,y)
```

```
3 4
```

Aber selbst wenn dieses interne Verhalten im Vergleich zu Programmiersprachen wie C, C ++, Perl oder Java seltsam erscheint, sind die beobachtbaren Ergebnisse der vorherigen Zuweisungen das, was wir erwartet haben, d. H. Wenn Sie sich die ID-Werte nicht ansehen. Es kann jedoch problematisch sein, wenn wir veränderbare Objekte wie Listen und Wörterbücher kopieren.

Python erstellt nur echte Kopien, wenn dies erforderlich ist, d. H. Wenn der Benutzer, der Programmierer, dies ausdrücklich verlangt.

Wir werden Ihnen die wichtigsten Probleme vorstellen, die beim Kopieren veränderlicher Objekte wie Listen und Wörterbücher auftreten können.

Variablen, die ein Objekt gemeinsam nutzen

Auf dieser Seite erfahren Sie, wie Sie Objekte, insbesondere Listen, kopieren. Trotzdem müssen Sie Geduld üben. Wir möchten vielen Anfängern etwas zeigen, das wie eine Kopie aussieht, aber nichts mit Kopien zu tun hat.

```
Farben1 = ["red", "blue"]
Farben2 = Farben1
print(Farben1, Farben2)
```

```
['red', 'blue'] ['red', 'blue']
```

Beide Variablen verweisen auf dasselbe Listenobjekt. Wenn wir uns die Identitäten der Variablen `Farben1` und `Farben2` ansehen, können wir sehen, dass beide Verweise auf dasselbe Objekt sind:

```
print(id(Farben1), id(Farben2))
```

```
2372364690120 2372364690120
```

Im obigen Beispiel wird Farben1 eine einfache Liste zugewiesen. Diese Liste ist eine sogenannte “flache Liste”, da sie keine verschachtelte Struktur aufweist, d. H. Keine Unterlisten in der Liste enthalten sind. Im nächsten Schritt weisen wir Farben2 Farben2 zu.

Die Funktion id () zeigt uns, dass beide Variablen auf dasselbe Listenobjekt zeigen, d. H. Sie teilen dieses Objekt.

Wir haben zwei Variablennamen `Farben1` und `Farben2`, die wir als gelbe Ovale dargestellt haben. Das blaue Feld symbolisiert das Listenobjekt. Ein Listenobjekt besteht aus Verweisen auf andere Objekte. In unserem Beispiel verweist das Listenobjekt, auf das beide Variablen verweisen, auf zwei Zeichenfolgenobjekte, d. H. “rot” und “blau”. Jetzt müssen wir untersuchen, was passieren wird, wenn wir nur ein Element der Liste von `Farben` oder `Farben1` ändern.

Jetzt wollen wir sehen, was passiert, wenn wir `Farben2` ein neues Objekt zuweisen. Wie erwartet bleiben die Werte von `Farben1` unverändert. Wie in unserem Beispiel im Kapitel “Datentypen und Variablen” wurde ein neuer Speicherort für `Farben2` zugewiesen , da wir dieser Variablen eine völlig neue Liste zugewiesen haben, dh ein neues Listenobjekt.

```
Farben1 = ["rot", "blau"]
Farben2 = Farben1
print(id(Farben1), id(Farben2))
```

```
2372363879240 2372363879240
```

```
Farben2 = "grün"
print(Farben1)
```

```
['rot', 'blau']
```

```
print(Farben1)
```

```
['rot', 'blau']
```

```
print(Farben2)
```

```
grün
```

Wir haben der Variablen 'Farben2' ein neues Objekt zugewiesen. Im folgenden Code ändern wir das Listenobjekt intern, indem wir dem zweiten Element der Liste einen neuen Wert zuweisen.

```
Farben1 = ["rot", "blau"]
Farben2 = Farben1

Farben2[1] = "grün"
```

Mal sehen, was im vorherigen Code im Detail passiert ist. Wir haben dem zweiten Element von Farben2 einen neuen Wert zugewiesen, d. H. Dem Element mit dem Index 1. Viele Anfänger werden überrascht sein, da die Liste von Farben1 ebenfalls "automatisch" geändert wurde. Natürlich haben wir keine zwei Listen: Wir haben nur zwei Namen für dieselbe Liste!

Die Erklärung ist, dass wir der Variablen `Farben2` kein neues Objekt zugewiesen haben. Wir haben das Objekt, auf das `Farben2` verweist, intern geändert oder wie es normalerweise als" an Ort und Stelle "bezeichnet wird. Beide Variablen `Farben1` und `Farben` zeigen immer noch auf dasselbe Listenobjekt.

Kopieren von Listen

Wir sind endlich beim Thema Kopieren von Listen angekommen. Die Klasse `list` bietet zu diesem Zweck die Methode `copy` an. Obwohl der Name klar zu sein scheint, hat dieser Weg auch einen Haken.

Der Haken ist zu sehen, wenn wir die Hilfe für `copy` verwenden:

```
help(list.copy)
```

```
Help on method_descriptor:

copy(self, /)
    Return a shallow copy of the list.
```

Viele Anfänger übersehen hier das Wort "flach". `help` sagt uns, dass eine neue Liste mit der Methode `copy` erstellt wird. Diese neue Liste ist eine "flache" Kopie der ursprünglichen Liste.

Im Prinzip ist das Wort "flach" in dieser Definition unnötig oder sogar irreführend.

Zuerst sollten Sie sich daran erinnern, was eine Liste in Python ist. Eine Liste in Python ist ein Objekt, das aus einer geordneten Folge von Verweisen auf Python-Objekte besteht. Das Folgende ist eine Liste von Zeichenfolgen:

Die Liste, auf die sich die Variable `Vornamen` bezieht, ist eine Liste von Zeichenfolgen. Grundsätzlich ist das Listenobjekt nur das blaue Feld mit den Pfeilen, d. H. Den Verweisen auf die Zeichenfolgen. Die Zeichenfolgen selbst sind nicht Teil der Liste.

```
Vornamen = ['Kevin', 'Jamina', 'Lars', 'Maria']
```

Im vorherigen Beispiel ist die Liste der Vornamen `Vornamen` homogen, d. H. Sie besteht nur aus Zeichenfolgen, sodass alle Elemente den gleichen Datentyp haben. Sie sollten sich jedoch der Tatsache bewusst sein, dass sich die Referenzen einer Liste auf beliebige Objekte beziehen können. Die folgende Liste `was-auch-immer` ist eine so allgemeinere Liste:

Wenn eine Liste kopiert wird, kopieren wir die Referenzen. In unseren Beispielen sind dies die blauen Kästchen, auf die mit `Vornamen` und mit `was auch immer` verwiesen wird. Die Auswirkungen davon werden im folgenden Unterkapitel gezeigt.

Probleme beim Kopieren von Listen

Das Kopieren von Listen kann leicht missverstanden werden, und dieses Missverständnis kann zu unangenehmen Fehlern führen. Wir werden dies in Form einer etwas anderen Liebesgeschichte präsentieren.

Stellen Sie sich eine Person vor, die in der Stadt Konstanz lebt. Die Stadt liegt im Süden Deutschlands am Bodensee. Der Rhein, der in den Schweizer Alpen beginnt, fließt durch den Bodensee und verlässt ihn erheblich größer. Diese Person namens Swen lebt in der “Seestrasse”, einer der teuersten Straßen von Konstanz. Seine Wohnung hat einen direkten Blick auf den See und die Stadt Konstanz. Wir haben einige Informationen über Swen in die folgende Listenstruktur aufgenommen:

```
Person1 = ["Swen", ["Seestrasse", "Konstanz"]]
```

Eines schönen Tages trifft die gutaussehende Sarah Swen, ihren charmanten Prinzen. Um die Geschichte kurz zu machen: Liebe auf den ersten Blick und sie zieht bei Swen ein.

Jetzt liegt es an uns, einen Datensatz für Sarah zu erstellen. Wir sind faul und werden Swens Daten recyceln, weil sie jetzt mit ihm in derselben Wohnung leben wird.

Wir werden Swens Daten kopieren und den Namen in Sarah ändern:

```
Person2 = Person1.copy()
Person2[0] = "Sarah"
print(Person2)
```

```
['Sarah', ['Seestrasse', 'Konstanz']]
```

Sie leben lange Zeit in perfekter Harmonie und tiefer Liebe, sagen wir ein ganzes Wochenende. Sie merkt plötzlich, dass Swen ein Monster ist: Er befleckt die Butter mit Marmelade und Honig und noch schlimmer, er breitet seine abgenutzten Socken im Schlafzimmer aus. Es dauert nicht lange, bis sie eine entscheidende Entscheidung trifft: Sie wird ihn und die Traumwohnung verlassen.

Sie zieht in eine Straße namens Bücklestrasse. Ein Wohngebiet, das bei weitem nicht so schön ist, aber zumindest ist sie vom Monster entfernt.

Wie können wir diesen Schritt aus Python-Sicht arrangieren?

Die Straße kann mit `person2[1][0]` erreicht werden. Also setzen wir dies auf den neuen Ort:

```
Person2[1][0] = "Bücklestraße"
```

Wir können sehen, dass Sarah erfolgreich an den neuen Standort gezogen ist:

```
print(Person2)
```

```
['Sarah', ['Bücklestraße', 'Konstanz']]
```

Ist das das Ende unserer Geschichte? Wird sie Swen nie wieder sehen? Lassen Sie uns Swens Daten überprüfen:

```
print(Person1)
```

```
['Swen', ['Bücklestraße', 'Konstanz']]
```

Swen ist anhänglich. Sie kann ihn nicht loswerden! Dies könnte für einige ziemlich überraschend sein. Warum ist es so? Die Liste `Person1` besteht aus zwei Verweisen: Einer auf ein String-Objekt (" Swen ") und der andere auf eine verschachtelte Liste (die Adresse`['Seestrasse', 'Konstanz']`. Wenn wir `copy` verwendet haben, haben wir nur diese Referenzen kopiert. Dies bedeutet, dass sowohl `person1[1]` als auch `Person2[1]`auf dasselbe Listenobjekt verweisen. Wenn wir Ändern Sie diese verschachtelte Liste, sie ist in beiden Listen sichtbar.

deepcopy aus der Modulkopie

Eine Lösung für das beschriebene Problem bietet das Modul `copy`. Dieses Modul stellt die Methode "Deepcopy" bereit, die eine vollständige oder tiefe Kopie einer beliebigen Liste ermöglicht, d. H. Flache und andere Listen.

Wiederholen wir das vorherige Beispiel mit der Funktion `deepcopy`:

```
from copy import deepcopy
Person1 = ["Swen", ["Seestrasse", "Konstanz"]]

Person2 = deepcopy(Person1)
Person2[0] = "Sarah"
```

Danach sieht die Implementierungsstruktur folgendermaßen aus:

Wir können sehen, dass die verschachtelte Liste mit der Adresse ebenfalls kopiert wurde. Wir sind jetzt gut vorbereitet auf Sarahs flüchtigen Schritt.

```
Person2[1][0] = "Bücklestrasse"
```

Sie ist erfolgreich ausgezogen und hat Swen endgültig verlassen, wie wir im Folgenden sehen können:

```
print(Person1, Person2)
```

```
['Swen', ['Seestrasse', 'Konstanz']] ['Sarah', ['Bücklestrasse', 'Konstanz']]
```

Aus Gründen der Übersichtlichkeit stellen wir hier auch ein Diagramm zur Verfügung.

Mit der ID-Funktion können wir feststellen, dass die Unterliste kopiert wurde, da sich `id (Person1[1])`von `id (Person2[1])` unterscheidet. Eine interessante Tatsache ist, dass die Zeichenfolgen nicht kopiert werden: `Person1[0]` und `Person2[0]` verweisen auf dieselbe Zeichenfolge.

```
print(id(Person1[1]), id(Person2[1]))
```

```
2372365051464 2372365051848
```

12 Python3 Dictionaries

12.0.1 Dictionaries

Einführung

In den vorigen Kapiteln haben wir die sequentiellen Datentypen wie Listen, Strings und Tupel eingeführt. Nun wollen wir eine weitere Kategorie von Datentypen genauer untersuchen: *Mappings*. Der zentrale eingebaute Mapping-Typ von Python ist `dict`, das Dictionary. Ein Dictionary ist ein assoziatives Feld. Assoziative Felder werden in verschiedenen Programmiersprachen mit verschiedenen Namen versehen. So spricht man in Java und C++ von einer Map, in Perl und Ruby von einem Hash und wie in Python bezeichnen auch Smalltalk, Objective-C und C# assoziative Arrays als Dictionaries.

Ein Dictionary besteht aus Schlüssel-Objekt-Paaren. Zu einem bestimmten Schlüssel gehört immer ein Objekt. Man kann also die Schlüssel auf die Objekte "abbilden", daher der Kategorienname Mapping.

Dictionaries gehören zu den wichtigsten Datentypen von Python. Kaum ein Programm oder Skript kommt ohne Dictionaries aus. Wie Listen können Dictionaries leicht verändert werden, außerdem können sie beliebig wachsen und schrumpfen während der Laufzeit.

In diesem Kapitel geht es aber nicht nur um Dictionaries sondern am Ende beschäftigen wir uns auch noch mit dem Zusammenhang zwischen Listen und Dictionaries, d.h. wir zeigen wie man aus Dictionaries Listen erzeugt und umgekehrt, wie man aus bestimmten Listen Dictionaries erzeugen kann.

Beispiele für Dictionaries Unser erstes Beispiel ist ein Dictionary mit deutschen Städten und Einwohnerzahlen, die im Jahre 2019 mehr als eine halbe Million Einwohner hatten. Wir sehen, dass Dictionaries mit geschweiften Klammern geschrieben werden. Sie enthalten Schlüssel-Werte-Paare, die mit Kommata getrennt sind. Ein Schlüssel (englisch key) und sein zugehöriger Wert werden jeweils durch einen Doppelpunkt getrennt.

```
städte_einwohner = {"Berlin": 3_669_491,
                    "Hamburg": 1_847_253,
                    "München": 1_484_226,
                    "Köln": 1_087_863,
                    "Frankfurt am Main": 763_380,
                    "Stuttgart": 635_911,
                    "Düsseldorf": 621_877,
                    "Leipzig": 593_145,
                    "Dortmund": 588_250,
                    "Essen": 582_760,
                    "Bremen": 567_559,
                    "Dresden": 556_780,
                    "Hannover": 536_925,
                    "Nürnberg": 518_370}
```

Wir können auf den Wert für einen bestimmten Schlüssel zugreifen, indem wir diesen Schlüssel in Klammern nach dem Namen des Wörterbuchs angeben:

```
städte_einwohner["Stuttgart"]
```

```
635911
```

Was passiert, wenn wir versuchen, auf einen Schlüssel zuzugreifen, d. h. eine Stadt in unserem Beispiel, die nicht im Dictionary enthalten ist? Wir erheben einen KeyError:

```
städte_einwohner["Konstanz"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-7-9368e7398149> in <module>
----> 1 städte_einwohner["Konstanz"]

KeyError: 'Konstanz'
```

Eine häufig gestellte Frage ist, ob Dictionary-Objekte geordnet sind. Seit Python 3.7 ist garantiert, dass ein Dictionary die **Einfügereihenfolge** seiner Schlüssel-Wert-Paare bewahrt. Das bedeutet jedoch nicht, dass die Einträge nach Schlüssel oder Wert sortiert werden. In unserem Beispiel behält `städte_einwohner` die Reihenfolge bei, in der die Einträge erzeugt wurden:

```
städte_einwohner
```

```
{'Berlin': 3669491,
 'Hamburg': 1847253,
 'München': 1484226,
 'Köln': 1087863,
 'Frankfurt am Main': 763380,
 'Stuttgart': 635911,
 'Düsseldorf': 621877,
 'Leipzig': 593145,
 'Dortmund': 588250,
 'Essen': 582760,
 'Bremen': 567559,
 'Dresden': 556780,
 'Hannover': 536925,
 'Nürnberg': 518370}
```

Die bewahrte Einfügereihenfolge macht ein Dictionary trotzdem nicht zu einer positionsbasierten Sequenz. Ein Ausdruck wie `d[0]` bedeutet immer „Wert zum Schlüssel `0`“ und nicht „erstes Element“. Existiert dieser Schlüssel nicht, entsteht ein `KeyError`:

```
städte_einwohner[0]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-9-1c0bd97222ea> in <module>
----> 1 städte_einwohner[0]

KeyError: 0
```

Wir hatten gesehen, dass wir auch eine Ausnahme erheben, wenn wir einen nicht existierenden Städenamen verwenden. Es ist sehr einfach weitere Einträge einem Dictionary einzufügen. Wir zeigen dies am Beispiel der Stadt Konstanz:

```
städte_einwohner["Konstanz"] = 84911
```

Wir können sehen, dass Konstanz ans Ende der Liste eingefügt worden ist:

```
städte_einwohner
```

```
{'Berlin': 3669491,
 'Hamburg': 1847253,
 'München': 1484226,
 'Köln': 1087863,
 'Frankfurt am Main': 763380,
 'Stuttgart': 635911,
 'Düsseldorf': 621877,
 'Leipzig': 593145,
 'Dortmund': 588250,
 'Essen': 582760,
 'Bremen': 567559,
 'Dresden': 556780,
 'Hannover': 536925,
 'Nürnberg': 518370,
 'Konstanz': 84911}
```

Dies ist also eine Möglichkeit ein Dictionary während des Programmlaufes zu erweitern. Denkbar ist dann auch mit einem leeren Dicitonary zu beginnen. Ein leeres Dictionary wird mit einem geschweiften Klammernpaar `{}` oder dem Aufruf `dict()` erzeugt:

```
food = {}
```

```
{}
```

Alternativ kann man `food` auch so erzeugen:

```
food = dict()
```

Zu Ehren des Schutzheiligen von Python "Monty Python" erweitern wir nun dieses Dictionary um spezielle pythonische kulinarische Besonderheiten. Was wäre Python ohne "ham", "eggs", "cheese" und "spam"? Wir erweitern unser Dictionary nun inkrementell um diese Delikatessen:

```
food["ham"] = "yes"
food["eggs"] = "yes"
food["cheese"] = "yes"
food
```

```
{'ham': 'yes', 'egg': 'yes', 'spam': 'no', 'cheese': 'yes', 'eggs': 'yes'}
```

Haben wir da nicht noch etwas vergessen? Leider, denn wer mag schon Spam. Bisher haben wir als Wert "yes" verwendet. Bei "spam" werden wir nun "no" verwenden. Bitte beachte

auch, dass die Werte in einem Dictionary nicht eindeutig sein müssen, weshalb wir mehrmals "yes" verwenden konnten. Die Schlüssel sind jedoch eindeutig.

```
food["spam"] = "no"
food
```

```
{'ham': 'yes', 'egg': 'yes', 'spam': 'no', 'cheese': 'yes', 'eggs': 'yes'}
```

Im nächsten Beispiel kreieren wir ein einfaches Deutsch-Englisches Wörterbuch als Dictionary:

```
en_de = {"red" : "rot", "green" : "grün", "blue" : "blau", "yellow":"gelb"}
print(en_de)
```

```
{'red': 'rot', 'green': 'grün', 'blue': 'blau', 'yellow': 'gelb'}
```

Wie wäre es mit einem weiteren Dictionary, eines was von Deutsch nach Französisch übersetzt? Damit sind wir dann aber auch in der Lage von Englisch nach Französisch zu übersetzen, indem wir die beiden Dictionaries hintereinander schalten, wie wir im folgenden Beispiel demonstrieren. Mit Hilfe dieses Dictionarys können wir nun von Englisch nach Französisch übersetzen, obwohl wir gar kein Englisch-Französisch-Wörterbuch haben. So gibt uns de_fr[en_de["red"]] das französische Wort für "red", also "rouge". Dies geschieht in zwei Schritten: Erst wird von Python der Ausdruck en_de["red"] ausgewertet. Das Ergebnis der Auswertung "rot" wird dann im de_fr-Dictionary "nachgeschaut", d.h. de_fr["rot"] liefert nun "rouge" zurück:

```
en_de = {"red" : "rot", "green" : "grün", "blue" : "blau", "yellow":"gelb"}
print(en_de)
```

```
{'red': 'rot', 'green': 'grün', 'blue': 'blau', 'yellow': 'gelb'}
```

```
print(en_de["red"])
```

```
rot
```

```
de_fr = {"rot": "rouge", "grün": "vert", "blau": "bleu", "gelb":"jaune"}
print("The French word for red is: " + de_fr[en_de["red"]])
```

```
The French word for red is: rouge
```

In einem Dictionary können beliebige Python-Objekte als Werte verwendet werden. Für Schlüssel gilt die präzisere Anforderung, dass sie **hashbar** (*hashable*) sein müssen. Strings, Zahlen und viele Tupel sind beispielsweise hashbar. Listen und Dictionaries sind es nicht und können deshalb nicht direkt als Schlüssel verwendet werden.

"Unveränderlich" ist für Einsteiger eine nützliche Faustregel, aber technisch entscheidet die Hashbarkeit. Auch ein Tupel ist nur dann als Schlüssel geeignet, wenn seine enthaltenen Werte hashbar sind.

Ein Versuch mit einer Liste als Schlüssel führt zu einem `TypeError`:

```
dic = { [1,2,3]:"abc"}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-af2a40fe8efa> in <module>
----> 1 dic = { [1,2,3]:"abc"}

TypeError: unhashable type: 'list'
```

Tupel als Schlüssel sind in Ordnung, wie wir im folgenden Beispiel sehen:

```
dic = { (1,2,3): "abc", 3.1415: "abc"}
dic
```

```
{(1, 2, 3): 'abc', 3.1415: 'abc'}
```

Nun wollen wir unsere Beispiele mit Wörterbüchern für natürliche Sprachen noch ein wenig aufpeppen. Dazu definieren wir ein Dictionary von Dictionaries:

```
en_de = {"red" : "rot", "green" : "grün", "blue" : "blau", "yellow":"gelb"}
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu", "gelb":"jaune"}

dictionaries = {"en_de" : en_de, "de_fr" : de_fr }

print(dictionaries["de_fr"]["blau"])
```

```
bleu
```

Operatoren auf Dictionaries

Operator

Erklärung

len(d)

liefert die Anzahl aller im Dictionary enthaltenen Elemente, d.h. Schlüssel-Werte-Paare

del d[k]

Löschen des Schlüssels k zusammen mit seinem Wert

k in d

True, wenn es im Dictionary d einen Schlüssel k gibt

k not in d

True, wenn es im Dictionary d keinen Schlüssel k gibt

Beispiele: Das folgende Dictionary enthält den Morsecode.

```
morse = {
"A" : ".-",
"B" : "-...",
"C" : "-.-.",
"D" : "-..",
"E" : ".",
"F" : "..-.",
```

```
"G" : "--.",
"H" : "....",
"I" : "..",
"J" : ".---",
"K" : "-.-",
"L" : ".-..",
"M" : "--",
"N" : "-.",
"O" : "---",
"P" : ".--.",
"Q" : "--.-",
"R" : ".-.",
"S" : "...",
"T" : "-",
"U" : "..-",
"V" : "...-",
"W" : ".--",
"X" : "-..-",
"Y" : "-.--",
"Z" : "--..",
"0" : "-----",
"1" : ".----",
"2" : "..---",
"3" : "...--",
"4" : "....-",
"5" : ".....",
"6" : "-....",
"7" : "--...",
"8" : "---..",
"9" : "----.",
"." : ".-.-.-",
"," : "--..--"
}
```

Sie können obiges Dictionary als morsecode.py abspeichern, um die folgenden Beispiele leichter nachvollziehen zu können. Zuerst müssen wir das Dictionary wie folgt importieren:

```
from morsecode import morse
```

Die Anzahl der verschiedenen Zeichen für die unser Dictionary eine Abbildung in ein Morsezeichen besitzt, können wir mittels len() bestimmen:

```
len(morse)
```

```
38
```

Unser Dictionary enthält nur Großbuchstaben. “a” in morse liefert deshalb zum Beispiel False zurück:

```
"a" in morse
```

```
False
```

```
"A" in morse
```

```
True
```

```
"a" not in morse
```

```
True
```

pop() und popitem()

pop Bei Dictionaries arbeitet `pop()` schlüsselbasiert: `D.pop(k)` entfernt den Eintrag mit dem Schlüssel `k` und liefert dessen Wert zurück. Wird kein passender Schlüssel gefunden und kein Defaultwert angegeben, entsteht ein `KeyError`. `popitem()` entfernt dagegen ein ganzes Schlüssel-Wert-Paar; bei heutigen Python-Versionen ist dies das zuletzt eingefügte Paar (LIFO).

```
capitals = {"Österreich":"Wien", "Deutschland":"Berlin",
↪  "Niederlande":"Amsterdam"}
capital = capitals.pop("Deutschland")
```

```
print(capital)
```

```
Berlin
```

```
print(capitals)
```

```
{'Österreich': 'Wien', 'Niederlande': 'Amsterdam'}
```

```
capital = capitals.pop("Schweiz")
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-26-1cd938366ca0> in <module>
----> 1 capital = capitals.pop("Schweiz")

KeyError: 'Schweiz'
```

Der Versuch die Hauptstadt der Schweiz zu ermitteln führte im vorigen Beispiel zum Ausnahmefehler KeyError, weil es diesen Eintrag im Dictionary nicht gibt. Um solche Fehler zu vermeiden, gibt es einen eleganten Weg in Python. Die Methode pop() hat dazu einen zweiten optionalen Parameter, mit dem man einen Default-Wert für diesen Fall mitgeben kann:

```
capital = capitals.pop("Schweiz","unbekannt")
print(capital)
```

```
unbekannt
```

```
capitals.pop("Niederlande","unbekannt")
```

```
'Amsterdam'
```

```
capitals.pop("Niederlande","unbekannt")
```

```
'unbekannt'
```

popitem `popitem()` benötigt keine Argumente. Es entfernt und liefert ein `(Schlüssel, Wert)`-Paar als Tupel. Seit Python 3.7 gilt dabei LIFO-Reihenfolge: Das zuletzt eingefügte Paar wird zuerst entfernt. Bei einem leeren Dictionary wird ein `KeyError` ausgelöst:

```
capitals = {"Hessen":"Wiesbaden", "Saarland":"Saarbrücken",
↪  "Baden-Württemberg":"Stuttgart", "Rheinland-Pfalz":"Mainz",
↪  "Nordrhein-Westfalen":"Düsseldorf"}
(land, capital) = capitals.popitem()
print(land, capital)
```

```
Nordrhein-Westfalen Düsseldorf
```

```
pair = capitals.popitem()
print(pair)
```

```
('Rheinland-Pfalz', 'Mainz')
```

```
print(capitals)
```

```
{'Hessen': 'Wiesbaden', 'Saarland': 'Saarbrücken', 'Baden-Württemberg':
↪  'Stuttgart'}
```

Zugriff auf nicht existierende Schlüssel

Versucht man auf nicht existierende Schlüssel zuzugreifen, erhält man eine Fehlermeldung:

```
woerter = {"house" : "Haus", "cat":"Katze"}
woerter["car"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-33-98bca352453b> in <module>
      1 woerter = {"house" : "Haus", "cat":"Katze"}
----> 2 woerter["car"]

KeyError: 'car'
```

Man kann dies wie folgt absichern:

```
if "car" in woerter: print(woerter["car"])
```

```
if "cat" in woerter: print(woerter["cat"])
```

```
Katze
```

Eine weitere Methode auf die Werte über die Schlüssel zuzugreifen ist mit der Methode get() gegeben. Auch ihr kann als zweiter Parameter ein Default-Wert mitgegeben werden:

```
capitals = {"Sachsen":"Dresden", "Niedersachsen":"Hannover",
↪  "Brandenburg":"Potsdam"}
capital = capitals["Sachsen"]
print(capital)
```

```
Dresden
```

```
capital = capitals["Thüringen"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-37-aeb15a8a3909> in <module>
----> 1 capital = capitals["Thüringen"]

KeyError: 'Thüringen'
```

```
capital = capitals.get("Sachsen")
print(capital)
```

```
Dresden
```

```
capital = capitals.get("Thüringen")
print(capital)
```

```
None
```

```
capital = capitals.get("Thüringen","Erfurt")
print(capital)
```

```
Erfurt
```

Wichtige Methoden

Mit der Methode copy() kann man ein Dictionary kopieren:

```
w = woerter.copy()
woerter["cat"]="chat"
print(w)
```

```
{'house': 'Haus', 'cat': 'Katze'}
```

```
print(woerter)
```

```
{'house': 'Haus', 'cat': 'chat'}
```

Bei diesem Kopieren handelt es sich um eine flache (shallow) Kopie. Wenn es sich bei einem Wert um einen komplexen Datentyp, wie z.B. eine Liste oder ein anderes Dictionary handelt,

wirken sich Änderungen innerhalb eines solchen Wertes sowohl auf die Kopie als auch auf das Original aus.

 # -- coding: utf-8 --

```
trainings = { "course1":{"title":"Python Training Course for Beginners",
                         "location":"Frankfurt",
                         "trainer":"Steve G. Snake"},
              "course2":{"title":"Intermediate Python Training",
                         "location":"Berlin",
                         "trainer":"Ella M. Charming"},
              "course3":{"title":"Python Text Processing Course",
                         "location":"München",
                         "trainer":"Monica A. Snowdon"}
              }

trainings2 = trainings.copy()

trainings["course2"]["title"] = "Perl Training Course for Beginners"
print(trainings2)
```

```
{'course1': {'title': 'Python Training Course for Beginners', 'location':
↪  'Frankfurt', 'trainer': 'Steve G. Snake'}, 'course2': {'title': 'Perl
↪  Training Course for Beginners', 'location': 'Berlin', 'trainer': 'Ella M.
↪  Charming'}, 'course3': {'title': 'Python Text Processing Course', 'location':
↪  'München', 'trainer': 'Monica A. Snowdon'}}
```

Wenn wir uns die Ausgaben anschauen, sehen wir, dass der Wert von “title” von “course2” sich sowohl in trainings als auch in trainings2 geändert hat.

Alles funktioniert so, wie man sich eine Kopie vorstellt, wenn man einem Schlüssel einen komplett neuen Wert, also ein neues Objekt zuordnet:

```
trainings = { "course1":{"title":"Python Training Course for Beginners",
                         "location":"Frankfurt",
                         "trainer":"Steve G. Snake"},
              "course2":{"title":"Intermediate Python Training",
                         "location":"Berlin",
                         "trainer":"Ella M. Charming"},
              "course3":{"title":"Python Text Processing Course",
                         "location":"München",
                         "trainer":"Monica A. Snowdon"}
              }

trainings2 = trainings.copy()

trainings["course2"] = {"title":"Perl Seminar for Beginners",
                         "location":"Ulm",
                         "trainer":"James D. Morgan"}
print(trainings2["course2"])
```

```
{'title': 'Intermediate Python Training', 'location': 'Berlin', 'trainer': 'Ella
↪  M. Charming'}
```

Für diejenigen, die mehr über die tieferen Gründe für dieses Verhalten erfahren wollen,

empfehlen wir unser Kapitel über “Flaches und tiefes Kopieren”.

Der Inhalt eines Dictionary kann mittels der Methode clear() geleert werden. Das Dictionary wird dabei nicht gelöscht sondern wirklich nur geleert:

```
trainings.clear()
print(trainings)
```

```
{}
```

Update: Einhängen eines weiteren Dictionary

Mit der Methode update() kann man ein zweites Dictionary in ein Dictionary einhängen. Enthält das zweite Dictionary Schlüssel, die auch im ersten vorkommen, so werden diese mit den Werten des zweiten überschrieben.

```
w={"house":"Haus","cat":"Katze","red":"rot"}
w1 = {"red":"rouge","blau":"bleu"}
w.update(w1)
print(w)
```

```
{'house': 'Haus', 'cat': 'Katze', 'red': 'rouge', 'blau': 'bleu'}
```

Iteration über ein Dictionary

Es bedarf keiner Methode, um über die Schlüssel eines Dictionarys zu iterieren:

```
d = {"a":123, "b":34, "c":304, "d":99}
for key in d:
    print(key)
```

```
a
b
c
d
```

Man kann aber auch die Methode keys() benutzen, die einem speziell die Schlüssel liefert:

```
for key in d.keys():
    print(key)
```

```
a
b
c
d
```

Mit der Methode values() iteriert man direkt über die Werte:

```
for value in d.values():
    print(value)
```

```
123
34
304
99
```

Wenn man nur auf die Ergebnisse schaut, ist das äquivalent zu der folgenden Schleife:

```
for key in d:
    print(d[key])
```

```
123
34
304
99
```

Was die Implementierung und die Effizienz betrifft, ist die letzte Methode jedoch weniger effizient. Im Folgenden zeigen wir, dass der erste Weg deutlich schneller ist. Um das Folgende zu verstehen, sollte man sich mit %%timeit von ipython auskennen:

```
%%timeit  d = {"a":123, "b":34, "c":304, "d":99}
for key in d.keys():
    x=d[key]
    x
```

```
564 ns ± 54.7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
%%timeit  d = {"a":123, "b":34, "c":304, "d":99}
for value in d.values():
    x=value
    x
```

```
371 ns ± 36.4 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Zusammenhang zwischen Listen und Dictionaries

Wenn man eine Weile mit Listen und Dictionaries unter Python gearbeitet hat, kommt nahezu zwangsläufig der Punkt, dass man Listen in Dictionaries oder umgekehrt Dictionaries in Listen wandeln will bzw. muss. Es wäre nicht allzu schwer, direkt mit den bisher bekannten Funktionen sich neue Funktionen zu schreiben, die genau dies bewerkstelligen. Aber Python wäre nicht Python, wenn es hierfür nicht auch Hilfen zur Verfügung stellen würde.

Es bietet sich an, ein Dictionary der Art

als Liste von 2-Tupel darzustellen.

Listen oder Mengen aus Dictionaries generieren

Immer wieder kommt es vor, dass man ein Dictionary hat und man aus diesem Dictionary Mengen oder Listen erzeugen will. Betrachten wir dazu das folgende Dictionary mit englisch-deutschen Farben:

```
colours = {"red":"rot", "green":"grün", "blue":"blau", "yellow":"gelb"}
```

Nehmen wir nun an, dass wir daraus gerne die Menge aller Schlüssel, also alle englischen Farbadjektive extrahieren wollen. Dazu bietet die Klasse dict die Methode “keys” an. Zu dieser Methode sagt uns die help-Funktion help(dict.keys):

```
help(dict.keys)
```

```
Help on method_descriptor:

keys(...)
    D.keys() -> a set-like object providing a view on D's keys
```

Zu Deutsch bedeutet dies, dass D.Keys für ein Dictionary D ein Mengen-ähnliches Objekt zurückliefert, was uns eine “view” (Sicht) auf die Schlüssel (keys) von D liefert. Eine “view” ist ein spezielles Python-Objekt, was in diesem Fall, wie die Hilfe sagt, ein einer Menge ähnelndes Objekt erzeugt. Dieses Objekt ist aber ein Iterator, der seine Elemente aus dem Dictionary generiert, ohne dass eine Menge generiert wird. Es erfolgt also keine Kopie.

```
english_colours = colours.keys()
english_colours
```

```
dict_keys(['red', 'green', 'blue', 'yellow'])
```

Im folgenden Beispiel können wir sehen, dass sich diese View auch “ändert”, wenn das Dictionary geändert wurde:

```
colours["black"] = "schwarz"
english_colours
```

```
dict_keys(['red', 'green', 'blue', 'yellow', 'black'])
```

Möchte man aus dem dict_keys-Objekt eine “echte” Menge oder eine Liste erzeugen, so kann man dies mit den globalen Funktionen set und list bewerkstelligen:

```
english_colours_set = set(english_colours)
colours["white"] = "weiß"
```

```
english_colours
```

```
dict_keys(['red', 'green', 'blue', 'yellow', 'black', 'white'])
```

```
english_colours_set
```

```
{'black', 'blue', 'green', 'red', 'yellow'}
```

```
english_colours_list = list(english_colours)
english_colours_list
```

```
['red', 'green', 'blue', 'yellow', 'black', 'white']
```

Analog dazu kann man sich auch eine view für die Werte eines Dictionaries (dict_values) mittels der Methode values erzeugen:

```
german_colours = colours.values()
german_colours
```

```
dict_values(['rot', 'grün', 'blau', 'gelb', 'schwarz', 'weiß'])
```

Außerdem gibt es noch die methode items(), mit der man sich eine View der Schlüssel-Werte-Paare erzeugen kann:

```
pairs = colours.items()
pairs
```

```
dict_items([('red', 'rot'), ('green', 'grün'), ('blue', 'blau'), ('yellow',
↪  'gelb'), ('black', 'schwarz'), ('white', 'weiß')])
```

```
set(pairs)
```

```
{('black', 'schwarz'),
 ('blue', 'blau'),
 ('green', 'grün'),
 ('red', 'rot'),
 ('white', 'weiß'),
 ('yellow', 'gelb')}
```

```
list(pairs)
```

```
[('red', 'rot'),
 ('green', 'grün'),
 ('blue', 'blau'),
 ('yellow', 'gelb'),
 ('black', 'schwarz'),
 ('white', 'weiß')]
```

Semantisch gesehen beinhaltet eine Liste/View, die mittels der Methode items() aus einem Dictionary erzeugt wurde, den kompletten Informationsgehalt des ursprünglichen Dictionary. Allerdings gibt es keine Zugriffsfunktionen über die Schlüssel (Keys), d.h. die erste Komponente der 2-Tupel und der Werte (Values), also der zweiten Komponente der 2-Tupel. ### Listen in Dictionaries wandeln Auch wenn wir uns im folgenden ein wenig mit Essen beschäftigen, bleibt dies ein Python-Kurs und wird kein Kochkurs. Nehmen wir an, dass wir zwei Listen haben, von denen eine die Keys und die andere die Werte enthält:

```
gerichte = ["Pizza", "Sauerkraut", "Paella", "Hamburger"]
laender = ["Italien", "Deutschland", "Spanien", "USA"]
```

Nun wollen wir ein Dictionary erzeugen, dass einem Gericht ein Land zuordnet. (Sorry, dass wir hier die Vorurteile bedienen.) Dazu benötigen wir die Funktion zip(). Der Name zip wurde gewählt, weil zip "Reißverschluss" im Englischen bedeutet. Wie ein Reißverschluss nimmt zip zwei oder mehr Listen - genauer gesagt iterierbare Objekte - als Argumente und liefert einen Iterator zurück, der die i-ten Komponenten der Argumente zu Tupeln zusammenfasst.

In unserem kulinarischen Beispiel sieht das wie folgt aus:

```
gericht_land_iterator = zip(laender, gerichte)
gericht_land_iterator
```

```
<zip at 0x2e61378b108>
```

```
land_gericht = list(gericht_land_iterator)
land_gericht
```

```
[('Italien', 'Pizza'),
 ('Deutschland', 'Sauerkraut'),
 ('Spanien', 'Paella'),
 ('USA', 'Hamburger')]
```

In der Variablen land_gericht haben wir nun eine Liste, in der die Elemente 2-Tupel sind mit den Schlüsseln und den Werten. Logisch gesehen sind wir also einem Dictionary schon sehr nahe. Was wir aber eigentlich wollen, ist ein richtiges Python-Dictionary. Glücklicherweise gibt es auch hierfür eine Funktion. dict() nimmt als Argument eine Liste der obigen Form und wandelt sie in ein Dictionary um:

```
land_gericht_dict = dict(land_gericht)
print(land_gericht_dict)
```

```
{'Italien': 'Pizza', 'Deutschland': 'Sauerkraut', 'Spanien': 'Paella', 'USA':
↪  'Hamburger'}
```

Allerdings ist das obige Vorgehen äußerst ineffizient, da wir zuerst eine Liste mit Zweiertupel erzeugen und diese dann in ein Dictionary wandeln. Man kann “dict” auch direkt auf das Ergebnis von zip aufrufen:

```
gerichte = ["Pizza", "Sauerkraut", "Paella", "Hamburger"]
laender = ["Italien", "Deutschland", "Spanien", "USA"]
dict(zip(laender, gerichte))
```

```
{'Italien': 'Pizza',
 'Deutschland': 'Sauerkraut',
 'Spanien': 'Paella',
 'USA': 'Hamburger'}
```

Eine Frage stellt sich noch bezüglich der Funktion zip(). Was passiert, wenn eine der beiden Listen mehr Elemente als die andere hat? Ganz einfach, die überzähligen Elemente werden nicht verarbeitet, d.h. sie kommen nicht in die Ergebnisliste.

```
laender = ["Italien","Deutschland","Spanien","USA","Schweiz"]
gerichte = ["Pizza", "Sauerkraut", "Paella", "Hamburger"]
land_gericht = zip(laender,gerichte)
print(land_gericht)
```

```
<zip object at 0x000002E611828E48>
```

Die brennende Frage, was wohl das Nationalgericht der Schweiz ist, bleibt, zumindest was diesen Kurs betrifft unbeantwortet!

Alles in einem Schritt Normalerweise empfehlen wir nicht zu viele Schritte in eine Anweisung oder einen Ausdruck zu stecken, auch wenn dies eindrucksvoll ausschauen mag und der Code dadurch sehr klein und kompakt wird. Es ist häufig besser Zwischenschritte mit sprechenden Variablen einzubauen. Dadurch wird die Lesbarkeit und Verständlichkeit eines Programmes erhöht.

Unser vorheriges kulinarisches Dictionary könnten wir in einem Schritt schreiben:

```
country_specialities_dict = dict(zip(["pizza", "sauerkraut", "paella",
↪  "hamburger"], ["Italy", "Germany", "Spain", "USA"," Switzerland"]))
print(country_specialities_dict)
```

```
{'pizza': 'Italy', 'sauerkraut': 'Germany', 'paella': 'Spain', 'hamburger':
↪  'USA'}
```

Andererseits sind die Zwischenschritte im folgenden Code möglicherweise des Guten zu viel:

```
dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
countries = ["Italy", "Germany", "Spain", "USA"]
country_specialities_zip = zip(dishes,countries)
country_specialities_list = list(country_specialities_zip)
country_specialities_dict = dict(country_specialities_list)
print(country_specialities_dict)
```

```
{'pizza': 'Italy', 'sauerkraut': 'Germany', 'paella': 'Spain', 'hamburger':
↪  'USA'}
```

Wir erhalten das gleiche Ergebnis wie im Einzeiler.

Iteratoren werden verbraucht `zip()` liefert in Python 3 einen Iterator. Viele Iteratoren können nur einmal vollständig durchlaufen werden: Nach dem ersten Durchlauf sind ihre Elemente verbraucht. Wenn man die Daten mehrfach benötigt, kann man den Iterator beispielsweise mit `list(...)` materialisieren oder `zip(...)` erneut erzeugen.

Wir demonstrieren dies im folgenden kleinen Beispiel:

```
l1 = ["a","b","c"]
l2 = [1,2,3]
c = zip(l1, l2)
for i in c:
    print(i)
```

```
('a', 1)
('b', 2)
('c', 3)
```

```
for i in c:
    print(i)
```

Diesen Effekt kann man auch beobachten, wenn man list() benutzt:

```
l1 = ["a","b","c"]
l2 = [1,2,3]
```

```
c = zip(l1,l2)
z1 = list(c)
z2 = list(c)
print(z1)
```

```
[('a', 1), ('b', 2), ('c', 3)]
```

```
print(z2)
```

```
[]
```

Als Übung können Sie überlegen, was im folgenden Skript faul ist:

```
dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
countries = ["Italy", "Germany", "Spain", "USA"]
country_specialities_zip = zip(dishes,countries)
print(list(country_specialities_zip))
country_specialities_list = list(country_specialities_zip)
country_specialities_dict = dict(country_specialities_list)
print(country_specialities_dict)
```

```
[('pizza', 'Italy'), ('sauerkraut', 'Germany'), ('paella', 'Spain'),
↪  ('hamburger', 'USA')]
{}
```

Wenn man das Skript startet, sieht man, dass das Dictionary, was wir erzeugen wollen, leer ist.

13 Python3 Eingabe

13.0.1 Eingabe

Eingabe mit Input

Es gibt kaum Programme, die ohne jegliche Eingaben auskommen. Eingaben können über viele Wege erfolgen, so zum Beispiel aus einer Datenbank, von einem anderen Rechner im lokalen Netzwerk oder auch über das Internet. Die einfachste und wohl auch häufigste Eingabe erfolgt jedoch über die Tastatur. Für diese Form der Eingabe bietet Python die Funktion input(eingabeprompt).

Kommt es zum Aufruf der Funktion input während eines Programmlaufes, wird der Programmablauf solange gestoppt, bis die Benutzerin oder der Benutzer eine Eingabe über die Tastur tätigt und diese mit der Return-Taste abschließt. Damit der User auch weiß, was er einzugeben hat, wird der String des Parameters "eingabeprompt" ausgegeben, sofern ein solcher String existiert. Der Parameter "eingabeprompt" von input() ist optional.

Der Eingabestring des Benutzers wird von input() nicht interpretiert, d.h. input() liefert immer einen String zurück.

In der folgenden interaktiven Python-Shell können wir erkennen, dass die Eingabe immer ein String ist.

```
eingabe = input("Ihre Eingabe? ")
eingabe
```

```
Ihre Eingabe? 34
```

```
'34'
```

```
eingabe = input("Ihre Eingabe? ")
```

```
Ihre Eingabe? 3, 5, 8
```

```
eingabe
```

```
'3, 5, 8'
```

```
type(eingabe)
```

```
str
```

Meistens wird auf den Ergebnisstring von input sofort der entsprechende cast-Operator angewendet, um den für die Weiterverarbeitung benötigten Datentyp zu erhalten:

```
alter = int(input("Ihr Alter? "))
```

```
alter, type(alter)
```

```
Ihr Alter? 42
```

```
(42, int)
```

```
preis = float(input("Artikelpreis? "))

preis, type(preis)
```

```
Artikelpreis? 33.99
```

```
(33.99, float)
```

```
preis = float(input("Preis? "))
print(preis)
```

```
Preis? 327.99
327.99
```

Eine unangenehme Überraschung erlebt mancher Programmieranfänger, wenn er oder sie eine Liste einlesen will und diese entsprechend mittels „list'' umwandelt:

```
staedte = list(input("Liste? "))
print(staedte)
```

```
Liste? ["Berlin", "Ulm", "Frankfurt", "Stuttgart", "Freiburg"]
['[', '"', 'B', 'e', 'r', 'l', 'i', 'n', '"', ',', ' ', '"', 'U', 'l', 'm', '"',
↪  ',', ' ', '"', 'F', 'r', 'a', 'n', 'k', 'f', 'u', 'r', 't', '"', ',', ' ',
↪  '"', 'S', 't', 'u', 't', 't', 'g', 'a', 'r', 't', '"', ',', ' ', '"', 'F',
↪  'r', 'e', 'i', 'b', 'u', 'r', 'g', '"', ']']
```

Eigentlich ist es klar und logisch, was passiert ist. Python wandelt den String in eine Liste, indem jedes einzelne Zeichen zu einem Listenelement wird. Wir hätten aber gerne die einzelnen Städte, also “Berlin”, “Ulm”, “Frankfurt”, “Stuttgart”, und “Freiburg”, als Elemente der Ergebnisliste. Um dies zu erreichen müssen wir die Funktion eval benutzen. eval interpretiert die Eingabe und liefert den entsprechenden Datentyp zurück. Dies funktioniert nicht nur für Listen sondern auch für andere Datentypen wie beispielsweise Tupel und Dictionaries.

```
staedte = eval(input("Liste? "))
print(staedte)
```

```
Liste? ["Zürich", "Bern", "Basel", "Schaffhausen"]
['Zürich', 'Bern', 'Basel', 'Schaffhausen']
```

```
einwohnerzahl = eval(input("Städte und Einwohner: "))
einwohnerzahl
```

```
Städte und Einwohner: {"Köln": 1024373, "Frankfurt":687775, "Stuttgart":597939}
```

```
{'Köln': 1024373, 'Frankfurt': 687775, 'Stuttgart': 597939}
```

Unterschied zu Python2

Da die Verwendung von input unter Python2 immer wieder zu Programmierfehlern geführt hatten, wurde das Verhalten von input() in Python3 zu dem von raw_input geändert. raw_input gibt es nicht mehr in Python3. Will man eine Benutzereingabe evaluieren, wie dies input() in Python2 automatisch macht, muss man dies in Python3 expliziert durch Anwendung der Funktion eval gestalten.

Im folgenden Diagramm stellen wir die Unterschiede zwischen input, raw_input und der verschiedenen Python-Versionen schematisch dar:

14 Python3 Bedingte Anweisungen

14.0.1 Bedingte Anweisungen

Manche Entscheidungen sind unvermeidlich

Zu bestimmten Zeiten und gewissen Situationen sind Entscheidungen unvermeidlich, da trennen sich die Wege von Mann und Frau, wie man im Foto sehen kann. So kann man auch kaum ein sinnvolles Programm oder Skript schreiben, was ohne jede Verzweigung auskommt.

In einer Programmiersprache versteht man unter einer bedingten Anweisung, wie man Verzweigungen auch nennt, Codeteile, die unter einer Bedingung ausgeführt, werden. Liegt die Bedingung nicht vor, wird dieser Code nicht ausgeführt. Anders ausgedrückt: Eine Verzweigung legt fest, welcher von zwei (oder auch mehr) Programmteilen (Alternativen) in Abhängigkeit von einer (oder mehreren) Bedingungen ausgeführt wird. Bedingte Anweisungen und Verzweigungen werden in Programmiersprachen (ebenso wie die Schleifen) den Kontrollstrukturen zugerechnet, weil mit ihrer Hilfe ein Programm auf verschiedene Zustände, die sich aus Eingaben und Berechnungen ergeben, reagieren kann.

Im abschließenden Beispiel dieses Kapitels verwenden wir einen historischen Einkommensteuertarif von 2014 als reine Programmierübung. Das Beispiel dient nur dazu, Verzweigungen zu demonstrieren und ist **kein aktueller Steuerrechner**.

Die if-Anweisung

Die if-Anweisung wird benutzt, um den Programmablauf in einem Python-Programm zu steuern. Damit wird es möglich zur Laufzeit zu entscheiden, ob bestimmte Programmteile ausgeführt werden sollen oder nicht.

Die einfachste Form einer if-Anweisung in einem Python-Programm sieht folgendermaßen aus:

Der eingerückte Block wird nur dann ausgeführt, wenn die Bedingung “bedingung” zutrifft. Sprich, wenn sie logisch “true” ist.

Im folgenden Beispiel wird der Benutzer nach seiner Nationalität gefragt. Der eingerückte Block wird nur ausgeführt, wenn die Nationalität “french” ist. Sollte der Benutzer eine andere Nationalität eingeben, passiert nichts.

```
person = input("Nationalität? ")
if person == "french":
    print("Préférez-vous parler français?")
```

```
Nationalität? french
Préférez-vous parler français?
```

Bitte beachten Sie: Wird hier `FFrench"` eingegeben, passiert nichts, denn Stringvergleiche

unterscheiden Groß- und Kleinschreibung. Für Benutzereingaben, die unabhängig von der Schreibweise verglichen werden sollen, kann man beispielsweise `casefold()` verwenden:

```
if nationality.casefold() == "french":
    ...
```

Im nächsten Schritt erweitern wir die Bedingung zunächst mit `or`.

Der italienische Kollege würde sich jetzt möglicherweise benachteiligt fühlen, weil italienisch sprechende nicht berücksichtigt wurden. Wir erweitern das Programm um eine weitere if-Anweisung.

```
person = input("Nationalität? ")
if person == "french" or person == "French":
    print("Préférez-vous parler français?")
```

```
Nationalität? French
Préférez-vous parler français?
```

```
person = input("Nationalität? ")
if person == "french" or person == "French":
    print("Préférez-vous parler français?")
if person == "italian" or person == "Italian":
    print("Preferisci parlare italiano?")
```

```
Nationalität? italian
Preferisci parlare italiano?
```

Dieses kleine Skript hat einen Nachteil. Nehmen wir an, dass der Benutzer “french” als Nationalität eingibt. In diesem Fall wird der Block des ersten “if” ausgeführt. Anschließend prüft das Programm aber noch, ob das zweite “if” ebenfalls ausgeführt werden kann. Das passiert hier natürlich nicht, denn die Eingabe “french” passt nicht zu der Bedingung “italian” oder “Italian”. Im Endeffekt ist dies eine unnötige Prüfung, wenn “french” oder “French” eingegeben wird.

Wir lösen das Problem mit einer “elif”-Bedingung. Der Ausdruck wird nur geprüft, wenn der Ausdruck des vorherigen “elif” oder “if” false war.

```
person = input("Nationalität? ")
if person == "french" or person == "French":
    print("Préférez-vous parler français?")
elif person == "italian" or person == "Italian":
    print("Preferisci parlare italiano?")
else:
    print("Du bist weder Italiener noch Franzose.")
    print("Deshalb reden wir jetzt deutsch!")
```

```
Nationalität? Türkisch
Du bist weder Italiener noch Franzose.
Deshalb reden wir jetzt deutsch!
```

Wie in unserem Beispiel gibt es in den meisten if-Anweisungen ebenfalls “elif”- und “else”-Zweige. Sprich, es kann mehr als einen “elif”-Zweig geben, aber nur einen “else”-Zweig. Der

“else”-Zweig muss am Ende der if-Anweisung stehen. Weitere “elif”-Zweige können nicht hinten angestellt werden.

Beispiel Hundejahre

Für Kinder und Hundeliebhaber ist es eine interessante und häufig gestellte Frage, wie alt ihr Liebling wohl wäre, wenn er kein Hund sondern ein Mensch wäre. Zur Berechnung dieser Aufgabe gibt es verschiedene wissenschaftliche und pseudowissenschaftliche Ansätze. Landläufig rechnet man Hundejahre in Menschen-Jahre um, indem man das Alter des Hundes mit 7 multipliziert. Je nach Hundegröße und Rasse sieht die Umrechnung jedoch etwas komplizierter aus, z.B.:

- Ein einjähriger Hund entspricht in etwa einem 14-jährigen Menschen
- Ein Hund der zwei Jahre alt ist, entspricht entwicklungsmäßig einem 22-jährigen Menschen.
- Jedes weitere Hundejahr entspricht fünf Menschenjahre.

Das folgende kleine Beispielprogramm in Python verlangt die Eingabe für das Alter des Hundes und berechnet nach obiger Regel das Alter in Menschenjahren:

```
age = int(input("Alter des Hundes: "))
print()
if age < 0:
    print("Das stimmt wohl kaum!")
elif age == 0:
    print("Entspricht 0 Menschenjahre!")
elif age == 1:
    print("Entspricht ca. 14 Jahre!")
elif age == 2:
    print("Entspricht ca. 22 Jahre!")
elif age > 2:
    human = 22 + (age -2) * 5
    print("Das entspricht ca. " + str(human) + " Menschenjahre!")
```

```
Alter des Hundes: 5

Das entspricht ca. 37 Menschenjahre!
```

Wir werden obiges Programm in unserem Python-Tutorial noch einmal im Kapitel “Funktionen” aufgreifen, um daraus eine Funktion zu machen.

Das obige Programm kann man auch mit einer zusätzlichen Variable schreiben, so dass das print nur einmal am Schluss des Programmes erscheint. Das ist stilistisch besser:

```
dog_age = int(input("Alter des Hundes: "))
print()
if dog_age <= 0:
    human_age = -1
elif dog_age == 1:
    human_age = 14
else:
    # this means: dog_age >= 2:
    human_age = 22 + (dog_age - 2) * 5
```

```
if human_age > 0:
    print("Entspricht " + str(human_age) + " Menschenjahren!")
else:
    print("Negative Werte oder Null ergeben keinen Sinn!")
```

```
Alter des Hundes:  6


Entspricht 42 Menschenjahren!
```

Wahr oder falsch: Truthiness

Eine Bedingung in Python muss nicht ausdrücklich `True` oder `False` enthalten. Python bestimmt den Wahrheitswert eines Objekts nach festen Regeln.

Als falsch (*falsy*) gelten insbesondere:

- `None` und `False`,
- numerische Nullwerte wie `0`, `0.0` und `0j`,
- leere Sequenzen und Collections, zum Beispiel , `[]`, `()`, `{}` und `set()`,
- Instanzen benutzerdefinierter Klassen, deren `__bool__()` `False` liefert oder deren `__len__()` den Wert `0` liefert.

Andere Objekte gelten normalerweise als wahr (*truthy*). Deshalb kann man beispielsweise statt `if len(liste) > 0:` idiomatisch `if liste:` schreiben.

Damit erklären sich Bedingungen, die nur aus einem Variablennamen bzw. Ausdruck bestehen.

```
my_list = eval(input("Type in list: "))

my_list_with_strings = []
while my_list:
    element = my_list.pop(0)
    if type(element) != str:
        element = str(element)
    my_list_with_strings.append(element)

print(my_list_with_strings)
```

```
Type in list:  [3, 5, (34, 5)]

['3', '5', '(34, 5)']
```

Die Schleife läuft über die Liste my_list. In jedem Schleifendurchlauf wird das Listenelement mit dem Index 0 mittels my_list.pop(0) aus der Liste entnommen. Alle Elemente von my_list werden in die Liste rev_list als Strings übernommen. Die Schleife endet, sobald my_list keine Elemente mehr enthält. Eine leere Liste entspricht ja nach dem oben Gesagten einem Wahrheitswert False!

Ternäres if

C-Programmierer kennen folgende abgekürzte Schreibweise für das if-Konstrukt:

als Abkürzung für:

In Python müssen sich C-Programmierer, was diese Schreibweise betrifft, umgewöhnen. In Python formuliert man das vorige Beispiel wie folgt:

Eigentlich ist es natürlich mehr als nur eine abgekürzte Schreibweise für eine standard if-Anweisung. Man kann das ternäre if auch innerhalb eines Ausdrucks verwenden:

```
a = 17
b = 50
result = (21 if a < b else 7) * 2
result
```

```
42
```

Steuerrechner in Python

Ein praktisches Beispiel für bedingte Anweisungen stellt die Einkommenssteuerberechnung dar. Das zu versteuernde Einkommen (zvE) muss einer Tarifzone zugeordnet werden, wozu sich eine if-Anweisung anbietet. Der Steuerbetrag (StB) bei Einzelveranlagung wird nach den entsprechenden Formeln berechnet:1

```
Erste Zone (Grundfreibetrag): bis 8353,- € fällt keine Steuer an
Zweite Zone: zvE von 8.354 € bis 13.469 €
StB = (974,58 * y + 1400)*y
Für y gilt:
y = (zvE - 8004) / 10000
Dritte Zone: zvE von 13470 € bis 52881 €
StB = (228,74 * z + 2397)*z + 971
Für z gilt:
z = (zvE - 13469) / 10000
Vierte Zone: zvE von 52882 € bis 250730 €
StB = 0,42 * zvE - 8239
Fünfte Zone: zvE ab 250731 €
StB = 0,45 * zvE - 15761
```

Eine Python-Funktion zur Berechnung der Einkommenssteuer für 2014 aus dem Einkommentarif sieht nun wie folgt aus: (natürlich ohne Gewähr!!!)

```
def steuern2014(einkommen):
    """Berechnung der zu zahlenden Steuern im Jahr 2014
    fuer ein zu versteuerndes Einkommen von x"""
    if einkommen < 8354:
        steuer = 0
    elif einkommen <= 13469:
        y = (einkommen - 8004.0)/10000.0
        steuer = (974.58 * y + 1400) * y
    elif einkommen <= 52881:
        z = (einkommen - 13469.0)/10000.0
        steuer = (228.74 * z + 2397.0) * z + 971
    elif einkommen < 250731:
        steuer = einkommen * 0.42 - 8239
```

```
    else:
        steuer = einkommen * 0.45 - 15761
    return steuer
```

Bei Ehegattensplitting lassen sich die Steuern mit folgendem Aufruf berechnen:

Wenn Sie auch 2013 und 2014 Ihre Einkommenssteuer mit Python berechnen wollen, finden Sie hier die die geänderten Funktion zur Steuerberechnung: (Ohne Gewähr)

```
def steuern2013(einkommen):
    """Berechnung der zu zahlenden Steuern im Jahr 2013
    fuer ein zu versteuerndes Einkommen von x"""
    if einkommen < 8130:
        steuer = 0
    elif einkommen <= 13469:
        y = (einkommen - 8004.0)/10000.0
        steuer = ( 933.70 * y + 1400) * y
    elif einkommen <= 52881:
        z = (einkommen - 13469.0)/10000.0
        steuer = (228.74 * z +2397.0) * z + 1014
    elif einkommen < 250731:
        steuer = einkommen * 0.42 - 8196
    else:
        steuer = einkommen * 0.45 - 15718
    return steuer
```

```
def steuern2014(einkommen):
    """Berechnung der zu zahlenden Steuern im Jahr 2014
    fuer ein zu versteuerndes Einkommen von x"""
    if einkommen < 8354:
        steuer = 0
    elif einkommen <= 13469:
        y = (einkommen -8004.0)/10000.0
        steuer = (974.58 * y + 1400) * y
    elif einkommen <= 52881:
        z = (einkommen -13469.0)/10000.0
        steuer = (228.74 * z +2397.0)*z +971
    elif einkommen < 250731:
        steuer = einkommen * 0.42 - 8239
    else:
        steuer = einkommen * 0.45 - 15761
    return steuer
```

Footnotes

1 Corresponding German legislative text: Einkommensteuergesetz, § 52 Anwendungsvorschriften (41) § 32a Absatz 1 ist ab dem Veranlagungszeitraum 2010 in der folgenden Fassung anzuwenden: “(1) 1Die tarifliche Einkommensteuer bemisst sich nach dem zu versteuernden Einkommen. 2Sie beträgt vorbehaltlich der §§ 32b, 32d, 34, 34a, 34b und 34c jeweils in Euro für zu versteuernde Einkommen
1.
bis 8 004 Euro (Grundfreibetrag): 0; 2.
von 8 005 Euro bis 13 469 Euro: (912,17 * y + 1 400) * y; 3.
von 13 470 Euro bis 52 881 Euro: (228,74 * z + 2 397) * z + 1 038; 4.

von 52 882 Euro bis 250 730 Euro: 0,42 * x - 8 172; 5.
von 250 731 Euro an: 0,45 * x - 15 694.

An den Formeln zur Berechnung der Einkommenssteuer hat sich im Jahr 2013 nichts prinzipiell geändert. Im folgenden sind die Änderungen in Rot markiert:

1. bis 8 354 Euro (Grundfreibetrag): 0;
2. von 8 355 Euro bis 13 469 Euro: (912,17 * y + 1 400) * y;
3. von 13 470 Euro bis 52 881 Euro: (228,74 * z + 2 397) * z + 1 038;
4. von 52 882 Euro bis 250 730 Euro: 0,42 * x - 8 239;
5. von 250 731 Euro an: 0,45 * x - 15 761. 3”y” ist ein Zehntausendstel des 8 004 Euro übersteigenden Teils des auf einen vollen Euro-Betrag abgerundeten zu versteuernden Einkommens. 4”z” ist ein Zehntausendstel des 13 469 Euro übersteigenden Teils des auf einen vollen Euro-Betrag abgerundeten zu versteuernden Einkommens. 5”x” ist das auf einen vollen Euro-Betrag abgerundete zu versteuernde Einkommen. 6Der sich ergebende Steuerbetrag ist auf den nächsten vollen Euro-Betrag abzurunden.”

15 Struktureller Musterabgleich

15.0.1 Struktureller Musterabgleich

Einführung

Unter dem strukturellen Musterabgleich (englisch “structural pattern matching”) versteht man eine Technik der Programmierung, die es ermöglicht, komplexe Datenstrukturen auf eine prägnante und lesbarere Weise in Vergleichen anzuwenden. Der strukturelle Musterabgleich wird in verschiedenen Programmiersprachen verwendet, unter anderem auch in Python.

Mit der strukturellen Mustererkennung wird eine Datenstruktur (z.B. eine Liste oder ein Tupel) analysiert und nach einem bestimmten Muster durchsucht, welches man vorher definiert hat. Man kann dann unterschiedliche Aktionen ausführen, basierend auf den gefundenen Mustern. Dies kann oft auf eine prägnante und leicht verständliche Weise geschrieben werden, was die Lesbarkeit des Codes erhöht.

Im Allgemeinen kann die strukturelle Mustererkennung die Fehleranfälligkeit verringern und die Produktivität bei der Entwicklung erhöhen, indem sie die Syntax verkürzt und die Lesbarkeit verbessert.

Der “strukturelle Musterabgleich” oder “struktureller Mustervergleich” wurde mit Python-Version 3.10. neu eingeführt. Die Syntax für diese neue Funktionalität wurde in PEP 622 im Juni 2020 vorgeschlagen. Der strukturelle Musterabgleich, wie ihn Python implementiert, wurde von einer ähnlichen Syntax in Scala, Erlang und anderen Sprachen inspiriert. In ihrer einfachsten Form verhält sie sich wie die switch-Anweisung in C, C++ oder Java. Es gibt jedoch noch weitere Anwendungsfälle für diese Funktion, wie man in diesem Kapitel unseres Python-Kurses lernen kann. Wir werden lernen, dass es in Python möglich ist, ein Muster in seine Grundkomponenten zu zerlegen.

Bevor Sie mit diesem Kapitel fortfahren, müssen Sie sicherstellen, dass auf Ihrem System die Python-Version 3.10 oder höher läuft. Der einfachste Weg, dies herauszufinden, ist die Verwendung von version aus sys:

```
import sys
sys.version
```

```
'3.10.10 (main, Mar 21 2023, 18:45:11) [GCC 11.2.0]'
```

In Python wird der strukturelle Musterabgleich durch die match-Anweisung eingeführt.

Die match-Anweisung funktioniert, indem sie einen ausgewerteten Ausdruck (auch bekannt als “zu prüfender Ausdruck”) mit einem oder mehreren Mustern (“Patterns”) vergleicht. Ein Pattern ist ein spezieller Ausdruck, der einem bestimmten Datentyp entspricht oder eine komplexe Datenstruktur wie eine Liste oder ein Tupel enthält.

Schauen wir uns ein einfaches Beispiel für den strukturellen Musterverabgleich an. Wir schreiben eine Funktion, die auf eine ausgewählte Sprache reagiert, also ‘chosen_language’

ist der zu prüfende Ausdruch:

```
def greeting(language):
    chosen_language = language.capitalize()
    match chosen_language:
        case 'English':
            print('Hello')
        case 'German':
            print('Hallo!')

greeting('english')
```

```
Hello
```

In Pythons' strukturellem Musterabgleich wird der Unterstrich (_) als Platzhalter verwendet, um einen beliebigen Wert zu finden. Das bedeutet für den folgenden Code, dass der Fall mit dem '_' immer dann greift, wenn nicht 'English' oder 'German' eingegeben wird:

```
def greeting(language):
    chosen_language = language.capitalize()
    match chosen_language:
        case 'English':
            print('Hello')
        case 'German':
            print('Hallo!')
        case _:
            print(f"Hallo, ich kenne {chosen_language} nicht")

greeting('english')
greeting('french')
```

```
Hello
Hallo, ich kenne French nicht
```

Wie Sie wahrscheinlich wissen, wird Deutsch nicht nur in Deutschland, sondern auch in Österreich und Teilen der Schweiz gesprochen. Wir sollten nicht vergessen, dass auch in Liechtenstein und Luxemburg Deutsch, zusätzlich zu Französisch, gesprochen wird. Dieses Beispiel zeigt nun einen Anwendungsfall, in dem der strukturelle Musterabgleich die Dinge erleichtert:

```
def greeting(language):
    chosen_language = language.split()
    match chosen_language:
        case ['English']:
            print('Hello')
        case ['German']:
            print('Hallo!')
        case ['German', 'AT']:
            print(f'Servus!')
        case ['German', 'CH']:
            print(f'Grüezi!')
        case ['German', 'DE']:
            print(f'Hallo')
        case ['German', 'LU']:
            print(f'Gudden Dag')
```

```
        case ['German', 'LI']:
            print(f'Grüezi')

for lang in ['English', 'German LU', 'German CH']:
    greeting(lang)
```

```
Hello
Gudden Dag
Grüezi!
```

Ohne 'match' würde der Python-Code ähnlich wie der folgende Code aussehen:

```
def greeting(language):
    chosen_language = language.split()
    if len(chosen_language) == 1:
        if chosen_language == ['English']:
            print('Hello')
        elif chosen_language == ['German']:
            print('Hallo!')
    elif len(chosen_language) == 2:
        if chosen_language == ['German', 'AT']:
            print(f'Servus!')
        elif chosen_language == ['German', 'CH']:
            print(f'Grüezi!')
        elif chosen_language == ['German', 'DE']:
            print(f'Hallo')
        elif chosen_language == ['German', 'LU']:
            print(f'Gudden Dag')
        elif chosen_language == ['German', 'LI']:
            print(f'Grüezi')

for lang in ['English', 'German LU', 'German CH']:
    greeting(lang)
```

```
Hello
Gudden Dag
Grüezi!
```

Wir könnten den vorherigen Code noch ein klein wenig verbessern, aber es sollte klar sein, dass `match` in diesem Fall in seiner Klarheit dennoch überlegen ist.

Wir können unser Beispiel weiter verbessern: Was ist, wenn jemand eine unbekannte Sprache oder eine unbekannte Sprache plus eine Region wählt? Das ist sehr einfach. Anstatt ein String-Literal in unserem Muster zu verwenden, benutzen wir Variablennamen:

```
def greeting(language):
    chosen_language = language.split()
    match chosen_language:
        case ['English']:
            print('Hello')
        case ['German']:
            print('Hallo!')
        case [unknown_language]:
            print(f"So far, we don't know how to greet in {unknown_language}!")
        case ['German', 'AT']:
            print(f'Servus!')
```

```
        case ['German', 'CH']:
            print(f'Grüezi!')
        case ['German', 'DE']:
            print(f'Hallo')
        case ['German', 'LU']:
            print(f'Gudden Dag')
        case ['German', 'LI']:
            print(f'Grüezi')
        case [lang, region]:
            print(f"Sorry, we don't know {lang} in your region {region}!")

for lang in ['English', 'German LU', 'French', 'German CH', 'English CA']:
    greeting(lang)
```

```
Hello
Gudden Dag
So far, we don't know how to greet in Dutch!
So far, we don't know how to greet in French!
Grüezi!
Sorry, we don't know English in your region CA!
```

Wir stellen einen weiteren interessanten Anwendungsfall in Form eines imaginären Abenteuerspiels vor. Die Spieler könnten Aktionen als Strings in den verschiedenen Formaten schreiben, wie

- 'help'
- 'show inventory'
- 'go north'
- 'go south'
- 'drop shield'
- 'drop all weapons'

```
commands = ['go north','go west', 'drop potion', 'drop all weapons',
↪  'drop shield']
weapons = ['axe','sword','dagger']
shield = True
inventory = ['apple','wood','potion'] + weapons + ['shield']
for command in commands:
    match command.split():
        case ["help"]:
            print("""You can use the following commands:
            """)
        case ["go", direction]:
            print('going', direction)
        case ["drop", "all", "weapons"]:
            for weapon in weapons:
                inventory.remove(weapon)
        case ["drop", item]:
            print('dropping', item)
            inventory.remove(item)
        case ["drop shield"]:
            shield = False
        case _:
            print(f"Sorry, I couldn't understand {command!r}")
```

```
going north
```

```
going west
dropping potion
dropping shield
['apple', 'wood']
```

Noch ein Beispiel, in dem wir die Fakultätsfunktion definieren:

```
def factorial(n):
    match n:
        case 0 | 1:
            return 1
        case _:
            return n * factorial(n - 1)

for i in range(6):
    print(i, factorial(i))
```

```
0 1
1 1
2 2
3 6
4 24
5 120
```

16 Abenteuer Spiel Mit Strukturellem Pattern Matching

16.0.1 Ein Text-basiertes Abenteuerspiel

Einführung

16.0.2 Ein Text-Abenteuerspiel

Einführung

In diesem Abschnitt unseres Python-Kurses führen wir strukturelles Muster-Matching anhand eines faszinierenden Szenarios ein - eines hypothetischen textbasierten Abenteuerspiels.

Text-Abenteuerspiele, oft als interaktive Fiktion bezeichnet, sind eine Art von Computerspiel, das hauptsächlich auf textbasierten Beschreibungen und typisierten Sprachbefehlen für die Benutzereingabe basiert, anstatt auf Grafiken und Sound. Typische Benutzereingaben für die meisten dieser Spiele sehen so aus:

- 'help'
- 'show inventory'
- 'go north'
- 'go south'
- 'drop shield'
- 'drop all weapons'

Ein bemerkenswertes Beispiel für ein solches Spiel ist "Hack", ein textbasiertes Rollenspiel, das in den 1980er Jahren am Massachusetts Institute of Technology (MIT) entwickelt wurde und allgemein als "NetHack" bekannt ist.

Wir werden Python-Code-Schnipsel präsentieren, die Szenarien innerhalb eines textbasierten Abenteuerspiels simulieren, während wir auch bedeutende Aspekte und Merkmale des strukturellen Muster-Matchings erläutern.

```
command = input("What are you doing next? ")
action, object = command.split()
print(f"{action=}, {object=}")
```

```
What are you doing next?  take sword

action='take', object='sword'
```

Was passiert, wenn der Benutzer weniger oder mehr als 2 Wörter eingibt?

```
command = input("What are you doing next? ")
words = command.split()
no_of_words = len(words)
```

```
if no_of_words == 1:
    print(f"action without object: {action=}")
elif no_of_words == 2:
    print(f"{action=}, {object=}")
else:
    print(words)
```

```
What are you doing next?  throw spear

action='take', object='sword'
```

Jetzt mit struktureller Mustererkennung. Übereinstimmung mehrerer Muster:

```
command = input("What are you doing next? ")
match command.split():
    case [action]:
        print(f"action without object: {action=}")
    case [action, obj]:
        print(f"{action=}, {obj=}")
```

```
What are you doing next?  look

action without object: action='look'
```

16.0.3 Übereinstimmung spezifischer Muster:

```
command = input("What are you doing next? ")
match command.split():
    case ["quit"]:
        print("Goodbye!")
        quit_game()
    case ["look"]:
        print("You are in cold and dark room ....")
    case ["get", obj]:
        print(f"You grab the {obj}")
    case ["go", direction]:
        print(f"You will go {direction} if possible")
    case _:
        print("Sorry, I do not understand!")
```

```
What are you doing next?  get apple

You grab the apple
```

16.0.4 Übereinstimmung mehrerer Werte:

```
command = "drop apple armour shield"

match command.split():
    case ["drop", *objects]:
        for obj in objects:
            print(f"You drop a{'n' if obj[0] in 'aeiou' else ''} {obj}")
```

```
You drop an apple
You drop an armour
You drop a shield
```

Hinzufügen eines Platzhalters

Wenn man im folgenden Code eine Eingabe verwendet, die keinem der definierten Muster entspricht, wirft Python keine Ausnahme. Stattdessen wird der Code einfach fortgesetzt, ohne in einen der Fallblöcke einzutreten, wie im folgenden Python-Beispiel:

```
command = 'fight monster'
match command.split():
    case ["help"]:
        print("""You can use the following commands:
        """)
    case ["go", direction]:
        print('going', direction)
    case ["drop", "all", "weapons"]:
        for weapon in weapons:
            inventory.remove(weapon)
    case ["drop", item]:
        print('dropping', item)
```

Um unerwartete oder nicht übereinstimmende Eingaben angemessen zu behandeln, kann man einen Platzhalterfall hinzufügen und einen Unterstrich (_) als Platzhalter verwenden, um alle Eingaben abzufangen, die nicht zu den spezifischen Mustern passen.

```
commands = ['go north','go west', 'drop potion', 'drop all weapons',
↪  'drop shield', 'fight monster']
weapons = ['axe','sword','dagger']
shield = True
inventory = ['apple','wood','potion'] + weapons + ['shield']
for command in commands:
    match command.split():
        case ["help"]:
            print("""You can use the following commands:
            """)
        case ["go", direction]:
            print('going', direction)
        case ["drop", "all", "weapons"]:
            for weapon in weapons:
                inventory.remove(weapon)
        case ["drop", item]:
            print('dropping', item)
            inventory.remove(item)
        case ["drop shield"]:
            shield = False
        case _:
            print(f"Sorry, I couldn't understand {command!r}")
```

```
going north
going west
dropping potion
```

```
dropping shield
Sorry, I couldn't understand 'fight monster'
```

16.0.5 Verschiedene Muster mit gleichem Ergebnis

```
command = input("What are you doing next? ")

match command.split():
    # Other cases
    # .....
    case ["north"] | ["go", "north"]:
        print("Let's go north")
    case ["look"] | ["look", "around"]:
        print("You are in a cold and dark room ....")
    case ["get", obj] | ["pick", "up", obj] | ["pick", obj, "up"]:
        print(f"Picking up {obj}")
```

```
What are you doing next?  north

Let's go north
```

16.0.6 Erfassen übereinstimmender Teilmuster

```
command = input("What are you doing next? ")
match command.split():
    case ["go", ("north" | "south" | "east" | "west") as direction]:
        print(f"You go {direction}")
```

```
What are you doing next?  north
```

16.0.7 Patterns mit Bedingungen

Wir können auch Bedingungen zu einem Zweig einer strukturellen Mustererkennung hinzufügen. Zum Beispiel überprüfen, ob ein Ausgang in eine bestimmte Richtung eines Raums existiert.

```
command = input("What are you doing next? ")
possible_direction = ['north', 'south']
match command.split():
    case ["go", direction] if direction in possible_direction:
        print(f"So, let's go {direction}!")
```

```
What are you doing next?  north
```

17 Python3 Schleifen

17.0.1 Schleifen

Allgemeiner Aufbau einer Schleife

Schleifen wiederholen einen Codeblock. Python besitzt dafür zwei zentrale Anweisungen:

- `while` wiederholt einen Block, solange eine Bedingung wahr ist.
- `for` iteriert über die Elemente eines *Iterable*, zum Beispiel über eine Liste, einen String, eine Datei oder ein `range`-Objekt.

Bei einer `while`-Schleife wird die Bedingung vor jedem Durchlauf geprüft. Solange ihr Wahrheitswert `True` ist, wird der Schleifenkörper ausgeführt. Danach springt die Programmausführung zur erneuten Prüfung der Bedingung zurück.

In anderen Programmiersprachen unterscheidet man häufig zwischen zähler-, bedingungs- und sammlungskontrollierten Schleifen. Python besitzt keine C-artige Syntax wie `for (i=0; i <= n; i++)`. Zählerschleifen lassen sich aber sehr natürlich mit `for` und `range()` ausdrücken:

```
for i in range(0, n + 1):
    print(i)
```

Die Python-`for`-Schleife ist grundsätzlich eine Iterationsschleife: Sie entnimmt nacheinander Werte aus einem Iterable. Dadurch eignet sie sich sowohl für Zählbereiche als auch für Listen und andere Datenstrukturen.

Einfache Beispiele von Schleifen

Das folgende Skript, das wir in der interaktiven Shell direkt eintippen können, gibt die Zahlen von 1 bis 10 aus:

```
i = 1
while i <= 10:
      print(i)
      i += 1
```

```
1
2
3
4
5
6
7
8
```

```
9
10
```

Mit dem nächsten kleinen Python-Skript berechnen wir die Summe der Zahlen von 1 bis 100. In ähnlicher Form würde man es auch in C, C++ oder Java machen. Allerdings geht es in Python deutlich einfacher, wie wir im folgenden Kapitel sehen werden.

```
n = 100

sum_of_numbers = 0
i = 1
while i <= n:
    sum_of_numbers += i
    i += 1

result = "Summe von 1 bis " + str(n) + ": " + str(sum_of_numbers)
print(result)
```

```
Summe von 1 bis 100: 5050
```

Standardeingabe lesen

Bevor wir mit der `while`-Schleife weitermachen, betrachten wir kurz die Standard-Datenströme. Ein Python-Prozess besitzt normalerweise:

- `sys.stdin` – Standardeingabe,
- `sys.stdout` – Standardausgabe,
- `sys.stderr` – Standardfehlerausgabe.

Im Terminal ist `stdin` häufig mit der Tastatureingabe verbunden; `stdout` und `stderr` erscheinen normalerweise im Terminal. Die Streams können jedoch auch umgeleitet werden.

Das folgende Beispiel liest mit einer `while`-Schleife Zeichen für Zeichen aus `sys.stdin`, bis ein Zeilenumbruch erreicht wird:

```
import sys

text = ""
while True:
    c = sys.stdin.read(1)
    if c == "":       # EOF
        break
    text += c
    if c == "\n":
        break

print(f"Eingabe: {text}")
```

Eleganter kann man eine beliebige Eingabezeile von der Standardeingabe natürlich mit der Funktion input(prompt) einlesen.

```
name = input("Wie heißen Sie?\n")
```

```
print(name)
```

```
Wie heißen Sie?
Tux
Tux
```

Der `else`-Teil einer Schleife

Sowohl `while`- als auch `for`-Schleifen können in Python einen optionalen `else`-Zweig besitzen. Der Name ist zunächst ungewohnt: Der `else`-Block wird ausgeführt, wenn die Schleife **ohne `break`** endet. Bei `while` geschieht dies normalerweise, wenn die Schleifenbedingung falsch wird; bei `for`, wenn das Iterable vollständig durchlaufen wurde.

Wird die Schleife durch `break` verlassen, wird ihr `else`-Block übersprungen. Genau dadurch wird diese Konstruktion beispielsweise bei Suchschleifen nützlich.

```
while Bedingung:
    Anweisungen
else:
    # nur wenn kein break ausgeführt wurde
    Anweisungen
```

Vorzeitiger Abbruch einer while-Schleife

Normalerweise wird eine while-Schleife nur beendet, wenn die Bedingung im Schleifenkopf nicht mehr erfüllt ist. Mit break kann man aber eine Schleife vorzeitig komplett verlassen. Mit 'continue' beendet man lediglich einen Durchlauf, d.h. man kehrt zum Schleifenkopf, also zur Überprüfung der Bedingung, zurück. Im folgenden Beispiel, einem einfachen Zahlenratespiel, kann man erkennen, dass in Kombination mit einem break der else-Zweig durchaus sinnvoll sein kann. Nur wenn die while-Schleife regulär beendet wird, d.h. der Spieler die Zahl erraten hat, gibt es einen Glückwunsch. Gibt der Spieler auf, d.h. break, dann wird der else-Zweig der while-Schleife nicht ausgeführt.

```
import random
n = 20
to_be_guessed = random.randint(1,n)

guess = 0
while guess != to_be_guessed:
    guess = int(input("Neuer Versuch: "))
    if guess > 0:
        if guess > to_be_guessed:
            print("Zu gross")
        elif guess < to_be_guessed:
            print("Zu klein")
    else:
        print("Schade, dass du aufgibst!")
        break
else:
    print("Gratuliere, das war's")
```

```
Neuer Versuch: 1
```

```
Zu klein
Neuer Versuch: 5
Zu klein
Neuer Versuch: 8
Zu klein
Neuer Versuch: 9
Zu klein
Neuer Versuch: 12
Zu klein
Neuer Versuch: 18
Zu klein
Neuer Versuch: 19
Zu klein
Neuer Versuch: 20
Gratuliere, das war's
```

Das vorige Programm prüft nicht, ob die `Zahl` einen Sinn ergibt, d.h. ob die Zahl zwischen einer Untergrenze und einer Obergrenze liegt. Wir können unser Programm verbessern. Die Grenzen müssen entsprechend der Benutzereingabe angepasst werden:

```
import random
upper_bound = 20
lower_bound = 1
to_be_guessed = random.randint(lower_bound, upper_bound)
guess = 0
while guess != to_be_guessed:
    guess = int(input("Dein Versuch: "))
    if guess == 0:   # Aufgegeben
        print("Sorry that you're giving up!")
        break   # breche Schleife ab und gehe nicht zu "else"
    if guess < lower_bound or guess > upper_bound:
        print("Rateversuch außerhalb der Schranken!")
    elif guess > to_be_guessed:
        upper_bound = guess - 1
        print("Zahl ist zu groß!")
    elif guess < to_be_guessed:
        lower_bound = guess + 1
        print("Zahl ist zu klein!")
else:
    print("Gratulation. Das war's!")
```

```
Dein Versuch:  10

Zahl ist zu groß!

Dein Versuch:  10

Rateversuch außerhalb der Schranken!

Dein Versuch:  5

Zahl ist zu groß!

Dein Versuch:  3

Zahl ist zu groß!

Dein Versuch:  3
```

```
Rateversuch außerhalb der Schranken!

Dein Versuch:  2

Gratulation. Das war's!
```

Übungen

Übung 1: Hundealter nach Menschenalter In unserem Kapitel [Bedingte Anweisungen] (./python3_bedingte_anweisungen.php) hatten wir ein Programm zur Umrechnung des Alters eines Hundes in ein menschliches Alter.

Das war das Programm:

```
dog_age = int(input("Alter des Hundes: "))
print()
if dog_age <= 0:
    human_age = -1
elif dog_age == 1:
    human_age = 14
else:
    # this means: dog_age >= 2:
    human_age = 22 + (dog_age - 2) * 5

if human_age > 0:
    print("Entspricht " + str(human_age) + " Menschenjahren!")
else:
    print("Negative Werte oder Null ergeben keinen Sinn!")
```

Schreiben Sie eine Version mit einer while-Schleife, damit die Leute wiederholt Hundealter umwandeln können. 0 oder ein negativer Wert bedeutet, dass sie aufhören wollen.

Übung 2: Fakultäts-Rechner Schreiben Sie ein Python-Programm, das die Fakultät einer vom Benutzer eingegebenen Zahl mit Hilfe einer while-Schleife errechnet. Die Fakultät einer nichtnegativen ganzen Zahl n, in der Mathematik als n! bezeichnet, ist das Produkt aller positiven ganzen Zahlen von 1 bis n. Zum Beispiel ist 5! (gelesen als "5 Fakultät") ist 5 * 4 * 3 * 2 * 1, was 120 entspricht.

Übung 3: Passwort-Prüfer Schreiben Sie ein Python-Programm, das einen einfachen Passwort-Prüfer simuliert. Das Programm soll den Benutzer auffordern, ein Passwort einzugeben, und ihn so lange auffordern, bis er das richtige Passwort eingibt. Sobald das richtige Passwort eingegeben wurde, soll das Programm eine Erfolgsmeldung ausgeben.

Übung 4: Passwort-Prüfer – Fortsetzung Erweitern wir das vorangegangene Beispiel für die Kennwortüberprüfung, indem wir eine maximale Anzahl von Versuchen hinzufügen. Wenn der Benutzer die maximale Anzahl an Versuchen überschreitet, soll das Programm ihn aussperren.

Übung 5: Primzahl-Prüfer Schreiben Sie ein Python-Programm, das prüft, ob eine gegebene positive ganze Zahl eine Primzahl ist. Das Programm soll den Benutzer auffordern, eine Zahl einzugeben und dann in einer while-Schleife prüfen, ob die Zahl eine Primzahl ist.

Übung 6: Fibonacci-Folge Schreiben Sie ein Python-Programm, das die Fibonacci-Folge bis zu einer bestimmten Anzahl von Termen erzeugt. Die Fibonacci-Folge ist eine Reihe von Zahlen, bei der jede Zahl die Summe der beiden vorhergehenden ist. Die ersten beiden Zahlen in der Folge sind normalerweise 0 und 1.

Übung 7: Collatz Sequenz Bei dem Problem geht es um Zahlenfolgen, die nach einem einfachen Bildungsgesetz konstruiert werden:

- Beginne mit irgendeiner natürlichen Zahl n > 0.
- Ist n gerade, so nimm als nächstes n / 2.
- Ist n ungerade, so nimm als nächstes 3n + 1.
- Wiederhole die Vorgehensweise mit der erhaltenen Zahl.

Zum Beispiel ergibt sich mit der Startzahl `n = 15` die Folge

```
46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1
```

Die Folge tritt somit in einen Zyklus ein, in dem die Zahlen 4, 2, 1 ständig wiederholt werden.

Die Collatz-Vermutung lautet nun:

```
Die Zahlenfolge mündet immer in den Zyklus 4, 2, 1, egal, mit welcher positiven natürlicher
```

Diese Vermutung konnte man bislang weder beweisen noch widerlegen.

Mathematische Formulierung:

Die **Collatz-Vermutung** besagt, dass für eine beliebige positive ganze Zahl n die Folge a_k schließlich die Zahl 1 für eine gewisse Anzahl k erreicht, wenn a_0 auf n gesetzt wird.}

$$a_{k+1} = \begin{cases} \frac{a_k}{2} & \text{if } a_k \text{ is even} \\ 3a_k + 1 & \text{if } a_k \text{ is odd} \end{cases}$$

- Schreibe ein Programm, das die Folge einer Zahl n ausgibt.
- Wie lang ist die Folge für die Zahl 271114753?
- Schreibe ein Programm, das die Längen der Collatz-Folgen für die Zahlen von 1 bis 100 ausgibt.

Lösungen

Löung zu Übung 1

```
dog_age = 1
while dog_age > 0:
    dog_age = int(input("Alter des Hundes: (0 oder negative für Abbruch"))
    print()
    if dog_age <= 0:
        human_age = -1
    elif dog_age == 1:
        human_age = 14
```

```
    elif dog_age >= 2:
        human_age = 22 + (dog_age - 2) * 5

    print("Entspricht  " + str(human_age) + " Menschenjahren!")

print("Programm ist zu Ende!")
```

```
Alter des Hundes: (0 oder negative für Abbruch 5


Entspricht  37 Menschenjahren!

Alter des Hundes: (0 oder negative für Abbruch 2


Entspricht  22 Menschenjahren!

Alter des Hundes: (0 oder negative für Abbruch 0


Entspricht  -1 Menschenjahren!
Programm ist zu Ende!
```

Löung zu Übung 2

```
n = int(input("Ganze Zahl deren Fakultät berechnet werden soll: "))

factorial = 1
counter = 1

while counter <= n:
    factorial *= counter
    counter += 1

print(f"{n}! = {factorial}")
```

```
Ganze Zahl deren Fakultät berechnet werden soll:  20

20! = 2432902008176640000
```

Löung zu Übung 3

```
correct_password = "password123"

user_input = input("Eingabe des Passwortes: ")
while user_input != correct_password:
    print("Falsches Passwort. Probiere es nochmals.")
    user_input = input("Eingabe des Passwortes: ")


print("Willkommen. Das war das korrekte Passwort!")
```

```
Enter the password:  geheim

Incorrect password. Please try again.
```

```
Enter the password:  password123

Willkommen. Das war das korrekte Passwort!
```

Alternativ können wir die Programmieraufgabe auch wie im Folgenden lösen. Dabei haben wir die Input-Zeile nur einmal. Deswegen müssen wir aber die Variable user_input vor der while-Schleife vorbesetzen und die input-Zeile muss als erstes im Schleifenkörper stehen:

```
correct_password = "password123"

user_input = " "
while user_input != correct_password:
    user_input = input("Eingabe des Passwortes: ")
    print("Falsches Passwort. Probiere es nochmals.")


print("Willkommen. Das war das korrekte Passwort!")
```

```
Eingabe des Passwortes:  geheim

Falsches Passwort. Probiere es nochmals.

Eingabe des Passwortes:  password123

Falsches Passwort. Probiere es nochmals.
Willkommen. Das war das korrekte Passwort!
```

Löung zu Übung 4

```
correct_password = "password123"
max_attempts = 3
attempts = 0

while attempts < max_attempts:
    user_input = input("Passwort eingeben: ")
    if user_input == correct_password:
        print("Willkommen. Das war das korrekte Passwort!")
        break
    else:
        attempts += 1
        remaining_attempts = max_attempts - attempts
        if remaining_attempts > 0:
            print(f"Falsches Passwort. Noch {remaining_attempts}
            ↪  Versuche übrig")
        else:
            print("Das war's maximale Eingebeversuche erreicht.")
```

```
Passwort eingeben:  geheim

Falsches Passwort. Noch 2 Versuche übrig

Passwort eingeben:  to secret

Falsches Passwort. Noch 1 Versuche übrig

Passwort eingeben:  password123

Willkommen. Das war das korrekte Passwort!
```

Löung zu Übung 5

```
number = int(input("Positive ganze Zahl eingeben: "))

divisor = 1
is_prime = True
while divisor <= number // 2:
    divisor += 1
    if number % divisor == 0 and is_prime:
        is_prime = False
        break  # Schleife wird verlassen, weil Divisor gefunden wurde

# Display the result
if is_prime:
    print(number, "ist eine Primzahl.")
else:
    print(number, "ist keine Primzahl.")
```

```
Positive ganze Zahl eingeben:  11

11 ist eine Primzahl.
```

Löung zu Übung 6

```
n_terms = int(input("Anzahl der gewünschten Fibonacci-Folge-Glieder: "))

old, new = 0, 1
count = 0
while count < n_terms:
    print(old, end=" ")
    old, new = new, old + new
    count += 1

print()
```

```
Anzahl der gewünschten Fibonacci-Folge-Glieder:  20

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Löung zu Übung 7

Der Algorithmus für die Collatz-Sequenz funktioniert wie folgt:

- Man beginnt mit einer beliebigen positiven ganzen Zahl n.
- Wenn n gerade ist, teile es durch 2.
- Wenn n ungerade ist, multipliziere es mit 3 und addiere 1.
- Wiederhole den Vorgang mit dem berechneten Wert als neuem Wert von n und fahre fort, bis n 1 wird.

```
n = int(input("Enter a positive integer: "))

if n <= 0:
    print("Please enter a positive integer.")
else:
    print(f"Collatz sequence for {n}:")
    while n != 1:
```

```
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        print(n, end=', ')
```

```
Enter a positive integer:  29

Collatz sequence for 29:
88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1,
```

```
n = int(input("Enter a positive integer: "))
length_of_sequence = 0
if n <= 0:
    print("Please enter a positive integer.")
else:
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        length_of_sequence += 1
print(f"Length of sequence: {length_of_sequence}")
```

```
Enter a positive integer:  271114753

Length of sequence: 279
```

```
counter = 1
stop_value = 100
while counter <= stop_value:
    #print(f"Collatz sequence for {counter}:")
    n = counter
    length_of_sequence = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        #print(n, end=", ")
        length_of_sequence += 1
    print(f"{counter}: {length_of_sequence}", end=", ")
    counter += 1
print()
```

```
1: 0, 2: 1, 3: 7, 4: 2, 5: 5, 6: 8, 7: 16, 8: 3, 9: 19, 10: 6, 11: 14, 12: 9, 13:
↪  9, 14: 17, 15: 17, 16: 4, 17: 12, 18: 20, 19: 20, 20: 7, 21: 7, 22: 15, 23:
↪  15, 24: 10, 25: 23, 26: 10, 27: 111, 28: 18, 29: 18, 30: 18, 31: 106, 32: 5,
↪  33: 26, 34: 13, 35: 13, 36: 21, 37: 21, 38: 21, 39: 34, 40: 8, 41: 109, 42:
↪  8, 43: 29, 44: 16, 45: 16, 46: 16, 47: 104, 48: 11, 49: 24, 50: 24, 51: 24,
↪  52: 11, 53: 11, 54: 112, 55: 112, 56: 19, 57: 32, 58: 19, 59: 32, 60: 19, 61:
↪  19, 62: 107, 63: 107, 64: 6, 65: 27, 66: 27, 67: 27, 68: 14, 69: 14, 70: 14,
↪  71: 102, 72: 22, 73: 115, 74: 22, 75: 14, 76: 22, 77: 22, 78: 35, 79: 35, 80:
↪  9, 81: 22, 82: 110, 83: 110, 84: 9, 85: 9, 86: 30, 87: 30, 88: 17, 89: 30,
↪  90: 17, 91: 92, 92: 17, 93: 17, 94: 105, 95: 105, 96: 12, 97: 118, 98: 25,
↪  99: 25, 100: 25,
```

18 Python3 For Schleife

18.0.1 For-Schleifen

Einführung

Wie auch die while-Schleife ist die for-Schleife eine Kontrollstruktur, mit der eine Gruppe von Anweisungen (ein Block) wiederholt ausführt werden kann.

Eine for-Schleife in Python wird für die Iteration über eine Sequenz (wie eine Liste, ein Tupel, eine Zeichenkette oder einen Bereich) oder andere iterierbare Objekte verwendet. Sie ist eine grundlegende Kontrollstruktur in der Programmierung, die es Ihnen ermöglicht, einen Codeblock eine bestimmte Anzahl von Malen zu wiederholen oder durch die Elemente einer Sequenz zu iterieren.

Was ist also mit der while-Schleife? Brauchen wir eine andere Art von Schleife in Python? Können wir nicht alles mit der while-Schleife machen? Ja, wir können "for"-Schleifen in "while"-Schleifen umschreiben. Aber bevor wir weitermachen, möchten Sie zumindest ein Beispiel für eine for-Schleife sehen.

Syntax einer for-Schleife As we mentioned earlier, the Python for loop is an iterator based for loop. It steps through the items of lists, tuples, strings, the keys of dictionaries and other iterables. The Python for loop starts with the keyword "for" followed by an arbitrary variable name, which will hold the values of the following sequence object, which is stepped through. The general syntax looks like this:

```
for <variable> in <sequence>:
    <statements>
else:
    <statements>
```

A first example of a for loop:

```
performance_levels = ("Python für Programmieranfänger",
                      "Python für Programmiererfahrene",
                      "Python für Fortgeschrittene")

# Iterate through each level in the tuple
for level in performance_levels:
    # Print the current level
    print(level)
```

```
Python für Programmieranfänger
Python für Programmiererfahrene
Python für Fortgeschrittene
```

Allgemeines über for-Schleifen

Dieses Unterkapitel kann von Anfängerinnen und Anfängern getrost übersprungen werden, da es hier vor allen Dingen um allgemeine Unterschiede von for-Schleifen in verschiedenen Programmiersprachen geht.

Die Syntax der For-Schleifen unterscheiden sich in den verschiedenen Programmiersprachen. Ebenso ist die Semantik einer For-Schleife, also wie sie vom Compiler oder Interpreter zu verstehen bzw. auszuführen ist, von Programmiersprache zu Programmiersprache unterschiedlich. Die "klassische" numerische Schleife, wie sie C und C++ kennt, besitzt eine Schleifenvariable, die mit einem Startwert initialisiert wird und nach jedem Durchlauf des Schleifenkörpers verändert wird, d.h. meistens um einen bestimmten Wert (z.B. 1) erhöht oder vermindert wird, bis der definierte Zielwert erreicht ist. Man nennt diese Schleifenform auch Zählschleife, weil die Schleifenvariable und damit auch der Startwert, der Endwert und die Schrittweite numerisch sein müssen. Im Beispiel sehen wir eine for-Schleife in C, die die Zahlen von 1 bis 100 ausdruckt:

```
for( i = 1; i <= 100; i++)
   printf("i: %d\n", i);
```

Auch wenn Sie diese Schleifenform bereits in C oder einer anderen Sprache liebgewonnen haben, müssen wir Sie leider enttäuschen: Python kennt keine solche for-Schleife. Wohlgemerkt "keine solche" aber sehr wohl eine for-Schleife. Die in Python benutzte Art von For-Schleife entspricht der in der Bash-Shell oder in Perl verwendeten foreach-Schleife. Bei dieser Schleifenart handelt es sich um ein Sprachkonstrukt mit dessen Hilfe nacheinander die Elemente einer Menge oder Liste bearbeitet werden können. Dazu werden sie einer Variable zugewiesen.

Syntax der For-Schleife in Python

Im folgenden sehen wir die allgemeine Syntax der for-Schleife in Python. Sequenz steht für ein iterierbares Objekt, also beispielsweise eine Liste, ein Tupel oder ein Dictionary.

Wie bereits gesagt, dient in Python die For-Schleife zur Iteration über ein Sequenz von Objekten, während sie in vielen anderen Sprachen meist nur "eine etwas andere while-Schleife" ist.

Beispiel einer for-Schleife in Python:

```
languages = ["C", "C++", "Perl", "Python"]
for language in languages:
    print(language)
```

```
C
C++
Perl
Python
```

Der optionale else-Block ist etwas Besonderes in Python. Während Perl-Programmierern dieses Konstrukt vertraut ist, ist es für C und C++-Programmierer ein ungewöhnliches Konzept. Semantisch funktioniert der optionale else-Block der for-Anweisung wie der else-Block der while-Anweisung. Er wird nur ausgeführt, wenn die Schleife nicht durch eine

break-Anweisung abgebrochen wurde. Das bedeutet, dass der else-Block nur dann ausgeführt wird, wenn alle Elemente der Sequenz abgearbeitet worden sind.

Trifft der Programmablauf auf eine break-Anweisung, so wird die Schleife sofort verlassen und das Programm wird mit der Anweisung fortgesetzt, die der for-Schleife folgt, falls es überhaupt noch Anweisungen nach der for-Schleife gibt.

Üblicherweise befindet sich die break-Anweisung innerhalb einer Konditionalanweisung, wie im folgenden Beispiel:

```
edibles = ["ham", "spam","eggs","nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

```
Great, delicious ham
No more spam please!
Finally, I finished stuffing myself
```

```
edibles = ["ham","eggs","nuts"]
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        break
    print("Great, delicious " + food)
else:
    print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

```
Great, delicious ham
Great, delicious eggs
Great, delicious nuts
I am so glad: No spam!
Finally, I finished stuffing myself
```

Vielleicht ist unsere Abscheu vor dem Dosenfutter "spam" nicht so groß, dass wir sofort aufhören zu essen. In diesem Fall kommt die continue-Anweisung ins Spiel. In dem folgenden kleinen Skript benutzen wir continue, um mit dem nächsten Artikel der essbaren Artikel weiterzumachen. "continue" schützt uns davor, "spam" essen zu müssen:

```
edibles = ["ham", "spam", "eggs","nuts"]
spam = False
for food in edibles:
    if food == "spam":
        print("No more spam please!")
        spam = True
        continue
    print("Great, delicious " + food)
    # here can be the code for enjoying our food :-)
else:
    if spam:
```

```
        print("I didn't like the spam!")
    else:
        print("I am so glad: No spam!")
print("Finally, I finished stuffing myself")
```

```
Great, delicious ham
No more spam please!
Great, delicious eggs
Great, delicious nuts
I didn't like the spam!
Finally, I finished stuffing myself
```

Die range()-Funktion

In Python gibt es eine einfache Möglichkeit Zählschleifen zu simulieren. Dazu benötigt man die range()-Funktion. range() liefert einen Iterator, der Zahlen in einem bestimmten Bereich (range) bei Bedarf, - also beispielsweise in einer For-Schleife, - liefern kann. Bevor wie die allgemeine Syntax angeben, zeigen wir die einfachste Benutzung von range() in einem Beispiel:

```
range(10)
```

```
range(0, 10)
```

```
list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Obiges Beispiel zeigt, dass range(), wenn man es mit einem einzelnen Argument aufruft, einen Iterator liefert, der die Zahlen von 0 (inklusive) bis zu diesem Wert (exklusive) generieren kann. Um eine entsprechende Liste aus dem Iterator zu erzeugen, benutzt man den cast-Operator list(). range() kann aber auch mit zwei Argumenten aufgerufen werden:

Dann wird ein Iterator für alle ganzen Zahlen von begin (einschließlich) bis end (ausschließlich) geliefert. Beispiel:

```
range(4, 10)
```

```
range(4, 10)
```

```
list(range(4, 10))
```

```
[4, 5, 6, 7, 8, 9]
```

Mit einem optionalen dritten Argument kann man range() noch die Schrittweite mitgeben, wie wir im folgenden Beispiel sehen:

```
list(range(4, 50, 5))
```

```
[4, 9, 14, 19, 24, 29, 34, 39, 44, 49]
```

Das ganze geht natürlich auch rückwärts:

```
list(range(42, -12, -7))
```

```
[42, 35, 28, 21, 14, 7, 0, -7]
```

Besonders sinnvoll wird die range()-Funktion im Zusammenspiel mit der for-Schleife. Im nachfolgenden Beispiel bilden wir die Summe der Zahlen von 1 bis 100:

```
n = 100

s = 0
for counter in range(1, n+1):
    s += counter

print("Sum of 1 until " + str(n) + ": " + str(s) )
```

```
Sum of 1 until 100: 5050
```

Dies lässt sich nun ganz einfach im Zusammenspiel mit range und der eingebauten Funktion "sum" bewerkstelligen. Mit "sum" kann man die Elemente numerischer Listen oder Tupel addieren, d.h. Listen oder Tupel, die nur numerische Werte enthalten. die Summe der Zahlen von 1 bis n lässt sich also ganz einfach wie folgt berechnen:

```
sum(range(101))
```

```
5050
```

Beispiel: Berechnung der pythagoräischen Zahlen

Beweis des Satzes von Pythagoras Die meisten glauben, dass der Satz von Pythagoras von Pythagoras entdeckt worden war. Warum sonst sollte der Satz seinen Namen erhalten haben. Aber es gibt eine Debatte, ob dieser Satz nicht auch unabhängig von Pythagoras und vor allen Dingen bereits früher entdeckt worden sein könnte. Für die Pythagoräer - eine mystische Bewegung, die sich auf die Mathematik, Religion und die Philosophie begründete - waren die ganzen Zahlen, die den Satz des Pythagoras erfüllten, besondere Zahlen, die für sie heilig waren.

Heutzutage haben die Pythagoräischen Zahlen nichts mystisches mehr. Obwohl sie für manche Schülerin oder Schüler oder andere Personen, die mit der Mathematik auf Kriegsfuß stehen, immer noch so erscheinen mögen.

Ganz unromantisch gilt in der Mathematik: Drei natürliche Zahlen, welche die Gleichung

erfüllen, heißen pythagoräische Zahlen.

Das folgende Programm berechnet alle pythagoräischen Zahlen bis zu einer einzugebenden maximalen Zahl:

```
from math import sqrt
n = int(input("Maximale Zahl? "))
for a in range(1,n+1):
    for b in range(a,n):
        c_square = a**2 + b**2
```

```
        c = int(sqrt(c_square))
        if ((c_square - c**2) == 0):
            print(a, b, c)
```

```
Maximale Zahl? 10
3 4 5
6 8 10
```

Iteration über eine Liste mit range()

Falls man auf die Indexe einer Liste zugreifen möchte, scheint es keine gute Idee zu sein eine For-Schleife zur Iteration über die Liste zu nutzen. Man kann dann zwar alle Elemente erreichen, aber der Index eines Elementes ist nicht verfügbar. Aber es gibt eine Möglichkeit sowohl auf den Index als auch auf das Element zugreifen zu können. Die Lösung besteht darin range() in Kombination mit der len()-Funktion, die einem die Anzahl der Listenelemente liefert, zu benutzen:

```
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
for i in range(len(fibonacci)):
    print(i,fibonacci[i])
print()
```

```
0 0
1 1
2 1
3 2
4 3
5 5
6 8
7 13
8 21
```

Listen-Iteration mit Seiteneffekten

Falls man über eine Liste iteriert, sollte man vermeiden die Liste im Schleifenkörper (body) zu verändern. Was passieren kann, zeigen wir im folgenden Beispiel:

```
colours = ["red"]
for i in colours:
    if i == "red":
        colours += ["black"]
    if i == "black":
        colours += ["white"]
print(colours)
```

```
['red', 'black', 'white']
```

Am besten benutzt man eine Kopie der Liste, wie im nächsten Beispiel:

```
colours = ["red"]
for i in colours[:]:
    if i == "red":
```

```
        colours += ["black"]
    if i == "black":
        colours += ["white"]
print(colours)
```

```
['red', 'black']
```

Auch jetzt haben wir die Liste verändert, aber "bewusst" innerhalb des Schleifenkörpers. Aber die Elemente, die über die For-Schleife iteriert werden, bleiben unverändert durch die Iterationen.

Übungen mit for-Schleifen

Übung 1: Fibonacci-Zahlen Erzeugen und drucken Sie die ersten `n` Zahlen der Fibonacci-Folge mit Hilfe einer for-Schleife.

Übung 2: Rautenmuster Erstellen Sie ein Programm, das ein Rautenmuster mit Sternchen (*) ausgibt, wie im folgenden Beispiel:

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

Übung 3: Primzahlen Schreiben Sie ein Programm, das mit Hilfe einer for-Schleife alle Primzahlen zwischen 1 und 50 findet und ausgibt.

Übung 4: Fizz, Buzz, FizzBuzz Bei der nächsten Aufgabe geht es um ein Kinderspiel. "Fizz buzz" ist ein Gruppen-Wortspiel für Kinder, das ihnen etwas über die mathematische Division beibringen soll. Die spielenden Kinder zählen abwechselnd inkremental, wobei jede durch drei teilbare Zahl durch das Wort „Fizz“ und jede durch fünf teilbare Zahl durch das Wort „Buzz“ ersetzt wird.

Schreibe ein Programm welches die Zahlen von 1 bis 42 ausgibt, aber entsprechend des Kinderspieles für Vielfache von 3 "Fizz" und für Vielfache von 5 "Buzz" ausgibt Für Zahlen, die Vielfache von sowohl 3 als auch 5 sind, wird "FizzBuzz" gedruckt.

Übung 5: Ramanujan-Hardy-Zahl In dieser Übung geht es um die Ramanujan-Hardy-Zahl. Es gibt eine kleine Anekdote des Mathematikers G.H. Hardy, als er den indischen Mathematiker Srinivasa Ramanujan im Krankenhaus besuchte. Sie lautet wie folgt:

Ich erinnere mich, dass ich ihn einmal besuchte, als er krank in Putney lag. Ich war im Taxi mit der Nummer 1729 mitgefahren und bemerkte, dass mir die Nummer ziemlich langweilig vorkam und ich hoffte, dass dies kein schlechtes Omen sei. "Nein", antwortete er, "es ist

eine sehr interessante Zahl; sie ist die kleinste Zahl, die sich als Summe zweier Quadrate auf zwei verschiedene Arten ausdrücken lässt."

Aus diesem Grund ist 1732 auch als Ramanujan-Hardy-Zahl bekannt.

Können Sie dies mit einem Python-Programm überprüfen?

Aufgabe 6: 1729 ist die kleinste Zahl, die durch eine Loeschsche quadratische Form $a^2 + ab + b^2$ auf vier verschiedene Arten dargestellt werden kann, mit positiven ganzen Zahlen a und b .

Finde diese Zahl mit einem Python-Programm.

Aufgabe 7: Pascalsche Dreiecksbeziehung Schreibe ein Python-Programm, das die ersten n Zeilen des Pascalschen Dreiecks erzeugt und ausgibt. Das Pascalsche Dreieck ist ein mathematisches Zahlenmuster, bei dem jede Zahl die Summe der beiden Zahlen direkt über ihr ist. Die ersten Zeilen des Pascalschen Dreiecks sehen wie folgt aus:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

Dein Programm soll eine ganze Zahl n als Eingabe annehmen und die ersten n Zeilen des Pascalschen Dreiecks ausgeben.Man muss verschachtelte for-Schleifen verwenden, um jede Zeile zu berechnen und zu drucken.

Diese Übung kann recht anspruchsvoll sein, aber sie ist eine gute Möglichkeit, verschachtelte Schleifen und die Erzeugung von Mustern in Python zu üben.

Lösungen

Lösung zur Übung 1

```
n = int(input("Anzahl der zu erzeugenden Fibonacci-Zahlen: "))

# Die beiden ersten Fibonacci-Zahlen
fibonacci_sequence = [0, 1]

for i in range(2, n):
    next_number = fibonacci_sequence[-1] + fibonacci_sequence[-2]
    fibonacci_sequence.append(next_number)

# Ausgabe der ersten n Fibonacci Zahlen
print("Fibonacci Sequence:")
for number in fibonacci_sequence[:n]:
    print(number, end=" ")
```

```
Anzahl der zu erzeugenden Fibonacci-Zahlen:  20

Fibonacci Sequence:
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Lösung zur Übung 2

```
height = int(input("Höhe der zu erzeugenden Raute: "))

height_is_even = height % 2 == 0
middle = height // 2 if height_is_even else height // 2 + 1

# Erzeugen der ersten Hälfte der Raute:
spaces = "*"
for counter in range(0, middle):
    print(spaces.center(middle*2+1))
    spaces += "**"
# zweite Hälfte der Raute:
spaces = spaces[:-2]
if height_is_even:
    # letzte Zeile muss verdoppelt werden:
    print(spaces.center(middle*2+1))
while len(spaces) > 1:
    spaces = spaces[:-2]
    print(spaces.center(middle*2+1))
```

```
Höhe der zu erzeugenden Raute:  11
```

```
      *
     ***
    *****
   *******
  *********
***********
  *********
   *******
    *****
     ***
      *
```

Lösung zur Aufgabe 3

```
print("Primzahlen zwischen 1 und 50:")
for number in range(1, 51):
    if number <= 1:
        is_prime = False
    elif number <= 3:
        is_prime = True
    elif number % 2 == 0 or number % 3 == 0:
        is_prime = False
    else:
        is_prime = True
        i = 5
        while i * i <= number:
            if number % i == 0 or number % (i + 2) == 0:
                is_prime = False
                break
            i += 6
    if is_prime:
        print(number, end=" ")
```

```
Primzahlen zwischen 1 und 50:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

Lösung zur Aufgabe 4

```
for number in range(1, 43):
    # Prüfe, ob die Uahl ein Vielfaches von 3 und 5 ist:
    if number % 3 == 0 and number % 5 == 0:
        print("FizzBuzz")
    # Prüfe, ob die Zahl ein Vielfaches von 3 ist
    elif number % 3 == 0:
        print("Fizz")
    # Prüfe, ob die Zahl ein Vielfaches von 5 ist
    elif number % 5 == 0:
        print("Buzz")
    else:
        print(number)
```

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
Buzz
26
Fizz
28
29
FizzBuzz
31
32
Fizz
34
Buzz
Fizz
37
38
Fizz
```

```
Buzz
41
Fizz
```

Lösung zur Aufgabe 5

```python
import math

number = 1729
n = int(number ** (1/3))

cubes = {}
for i in range(n+1):
    for j in range(i):
        result = i ** 3 + j ** 3
        if result in cubes:
            cubes[result].append((i, j))
        else:
            cubes[result] = [(i,j)]
        if result > number:
            break

for x in cubes:
    if len(cubes[x]) > 1:
        print(x, cubes[x])
```

```
1729 [(10, 9), (12, 1)]
```

Lösung zur Aufgabe 6

```python
import math

number = 1729
n = int(number ** (1/2))

results = {}
for a in range(n+1):
    for b in range(a):
        result = a**2 + a*b + b**2
        if result in results:
            results[result].append((a, b))
        else:
            results[result] = [(a,b)]
        if result > number:
            break

for x in results:
    if len(results[x]) > 3:
        print(x, results[x])
```

```
1729 [(25, 23), (32, 15), (37, 8), (40, 3)]
```

Lösung zur Aufgabe 7

```
n = int(input("Zeilenanzahl Pascal'sches Dreieck: "))

triangle = []
# Erzeugung des Pascal'schen Dreiecks
for i in range(n):
    row = []
    for j in range(i + 1):
        if j == 0 or j == i:
            # rstes und letztes Element einer Reihe sind 1
            row.append(1)
        else:
            previous_row = triangle[i - 1]
            row.append(previous_row[j - 1] + previous_row[j])
    triangle.append(row)

# Ausgabe Pascal'sches Dreieck
for row in triangle:
    print(" ".join(map(str, row)).center(n * 3))
```

```
Zeilenanzahl Pascal'sches Dreieck:  9
```

```
            1
           1 1
          1 2 1
         1 3 3 1
        1 4 6 4 1
      1 5 10 10 5 1
     1 6 15 20 15 6 1
   1 7 21 35 35 21 7 1
  1 8 28 56 70 56 28 8 1
```

```

```

19 Python3 Walross Operator

19.0.1 Zuweisungsausdrücks alias Walross-Operator

Einführung

Die Python-Version 3.8 brachte eine neue Funktion mit sich, auf die einige Python-Programmierer schon lange gewartet hatten. Eingeführt wurde eine neue Möglichkeit Objekten Variablen zuzuweisen, nämlich der := Operator. Er bietet Programmierenden eine bequeme Möglichkeit, Variablen in der Mitte von Ausdrücken zuzuweisen. Wenn man sich die Zeichen mit ein wenig Phantasie ansieht, kann man eine Ähnlichkeit mit den Augen und Stoßzähnen eines Walrosses erkennen, weshalb dieser Operator auch liebevoll **der Walross-Operator** genannt wird.

Die Zuweisungsausdrücke wurden in [PEP 572] (https://www.python.org/dev/peps/pep-0572/) diskutiert, und dies ist, was über die Benennung geschrieben wurde:

> Während der Diskussion über dieses PEP wurde der Operator informell als "der Walross-Operator" bekannt. Der formale Name des Konstrukts ist Zuweisungsausdrücke (englisch "Assignment Expressions") (gemäß dem PEP-Titel), aber sie können auch als benamte Ausdrücke (englisch "Named Expressions") bezeichnet werden (z.B. verwendet die CPython-Referenzimplementierung intern diesen Namen)
>
> Englischer Originaltext: Wir werden diese Art der Zuweisung in diesem Teil unseres Python-Tutorials vorstellen.

Eine einfache Zuweisung kann auch durch einen Zuweisungsausdruck ersetzt werden, auch wenn dies unbeholfen aussieht und definitiv nicht der beabsichtigte Anwendungsfall ist:During discussion of this PEP, the operator became informally known as "the walrus operator". The construct's formal name is "Assignment Expressions" (as per the PEP title), but they may also be referred to as "Named Expressions" (e.g. the CPython reference implementation uses that name internally).

Wir werden diese Art der Zuweisung in diesem Teil von unserem Python-Kurse vorstellen.

Eine einfache Zuweisung kann auch durch einen Zuweisungsausdruck ersetzt werden, auch wenn dies unbeholfen aussieht und definitiv nicht der beabsichtigte Anwendungsfall ist. Man sollte unbedingt beachten, dass die Klammern in unten stehendem Beispiel unbedingt notwendig sind:

```
x = 5
# kann auch wie folgt geschrieben werden:
(x := 5)  # gültig, aber nicht empfohlen
```

```
5
```

Schauen wir uns nun ein kleines Code-Beispiel an, das nur traditionelle Zuweisungen

verwendet:

```
txt = 'Python-lernen macht Spaß!'
ideal_length = 22

n = len(txt)
if n == ideal_length:
    print(f'Die Länge {n} ist ideal!')
else:
    print(f'Die Länge {n} ist nicht ideal!')
```

```
Die Länge 25 ist nicht ideal!
```

Wie schreiben obiges Python-Beispiel um, indem wir den neuen Walross-Operator verwenden:

```
txt = 'Python-lernen macht Spaß!'
ideal_length = 22

if (n := len(txt)) == ideal_length:
    print(f'Die Länge {n} ist ideal!')
else:
    print(f'Die Länge {n} ist nicht ideal!')
```

```
Die Länge 25 ist nicht ideal!
```

Okay, einige werden vielleicht sagen, dass dies nicht sehr beeindruckend ist und die erste Version sogar noch lesbarer sei. Deshalb wollen wir nun einen Blick auf einen nützlicheren Anwendungsfall werfen.

Nützliche Anwendungen der Zuweisungsausdrücke

Listen-Abstraktion (list comprehension) Im Folgenden benutzen wir den Walross-Operator in einer Listenabstraktion:

```
def f(x):
    return x + 4

numbers = [3, 7, 2, 9, 12]

odd_numbers = [result for x in numbers if (result := f(x)) % 2]
odd_numbers
```

```
[7, 11, 13]
```

Die obige Implementierung ist effizienter als eine Listenabstraktion ohne den Zuweisungsausdruck, da wir die Funktion nämlich zweimal aufrufen müssen:

```
def f(x):
    return x + 4

numbers = [3, 7, 2, 9, 12]

odd_numbers = [f(x) for x in numbers if  f(x) % 2]
```

```
odd_numbers
```

```
[7, 11, 13]
```

Reguläre Ausdrücke Bei der Verwendung regulärer Ausdrücke birgt der Walross-Operator ebenfalls einen großen Vorteil. Der Textstring macht keinen großen Sinn. Es geht um Anfangs- und Enddaten von Python-Kursen. Diese Datumsangaben kommen in verschiedenen Formaten vor:

```
import re

txt = """Die Python-Schulung begann am 2023-03-20
sie endete am 2023-03-24
nur ein Datum pro Zeile, wenn überhaupt
die Daten können auch in diesem Format sein 2023/03/15
oder 23-02-04"""

for line in txt.split('\n'):
    if (date := re.search(r'(\d{2,4})[-/](\d{2})[-/](\d{2})', line)):
        year, month, day = date.groups()
        print(year, month, day)
```

```
2023 03 20
2023 03 24
2023 03 15
23 02 04
```

Lesen von Dateien mit fester Zeilenlänge Die Datei 'python_kurse_daten.txt' enthält Zeilen fester Länge, d.h. jede Zeile ist 57 Zeichen lang. Jede Zeile enthält einen Kurstitel und Daten für den Beginn und das Ende des Kurses. Die Datei sieht wie folgt aus:

```
Python-Kurs: Basics                2023-06-19 2023-06-23
Java-Kurs: Basics                  2023-06-26 2023-06-30
Python-Kurs: Machine Learning      2023-07-03 2023-07-07
Python-Kurs: OOP                   2023-07-24 2023-07-28
```

Der folgende Python-Code schält diese INformationen aus den Zeilen. Wir lesen jeweils 57 Zeichen ein und weisen das Ergebnis mittels Walross-Operator der Variablen data zu:

```
with open('python_kurse_daten.txt') as fh:
    while ((data := fh.read(57)) != ''):
        kurs = data[:35].rstrip()
        anfang = data[35:46]
        ende = data[46:].rstrip()
        print(kurs)
        print(anfang)
        print(ende)
```

```
Python-Kurs: Basics
2023-06-19
2023-06-23
```

```
Java-Kurs: Basics
2023-06-26
2023-06-30
Python-Kurs: Machine Learning
2023-07-03
2023-07-07
Python-Kurs: OOP
2023-07-24
2023-07-28
```

Weiteres Beispie mit while-Schleife Im Kapitel über while-Schleifen unseres Python-Tutorials haben wir ein kleines Zahlen-Ratespiel vorgestellt:

```
import random

lower_bound, upper_bound = 1, 20
to_be_guessed = random.randint(lower_bound, upper_bound)
guess = 0
while guess != to_be_guessed:
    guess = int(input("Neue Zahl: "))
    if guess > to_be_guessed:
        print("Zahl zu groß")
    elif guess < to_be_guessed:
        print("Zahl zu klein")
else:
    print("Glückwunsch! Du hast es geschafft!")
```

Wie man sehen kann, mussten wir 'guess' auf Null initialisieren, um in die Schleife einsteigen zu können. Wir können die Initialisierung direkt in der Schleifenbedingung mit einem Zuweisungsausdruck durchführen und so den gesamten Code vereinfachen:

```
import random

lower_bound, upper_bound = 1, 20
to_be_guessed = random.randint(lower_bound, upper_bound)

while (guess := int(input("New number: "))) != to_be_guessed:
    if guess > to_be_guessed:
        print("Number too large")
    elif guess < to_be_guessed:
        print("Number too small")
else:
    print("Congratulations. You made it!")
```

```
New number:  10

Number too small

New number:  15

Congratulations. You made it!
```

Auf die Spitze getrieben:

Wir haben am Anfang dieser Seite gesagt, dass einige Python-Programmierer dieses Konstrukt schon lange herbeigesehnt haben. Ein Grund, warum es nicht früher eingeführt wurde, war die Tatsache, dass man damit auch Code schreiben kann, der weniger lesbar ist, wenn man dies Möglichkeit zu extensiv verwendet. Der folgende Codeschnipsel zeigt ein solches Extrembeispiel, dessen Verwendung nicht zu empfehlen ist:

```
a, b, c = 1, 2, 3
x = 4
y = (c := (a := x*2.3) + (b := x*4.5 -3))
```

```
y
```

```
24.2
```

```
c
```

```
24.2
```

```
a
```

```
9.2
```

20 Zuweisungsausdrücke

20.0.1 Zuweisungsausdrücke, auch bekannt als Walrus-Operator.

Einführung

Dieser Abschnitt in unserem Python-Kurs erkundet den **“Zuweisungsoperator”**, den viele liebevoll als **“den Walross-Operator”** bezeichnen. Man kann ihn also entweder als den **“Walross-Operator”** oder als den **“Zuweisungsoperator”** bezeichnen, und die Menschen in der Python-Programmiergemeinschaft werden im Allgemeinen verstehen, was man meint. Obwohl er erst in Python-Version 3.8 im Oktober 2019 eingeführt wurde, können wir ihn als eine relativ neue Ergänzung zur Sprache betrachten. Die Zuweisungsausdrücke wurden in “PEP 572”1 diskutiert, und das ist das, was über die Benennung geschrieben wurde:

```
During discussion of this PEP, the operator became informally known as "the walrus operator
```

Wir werden den Zuweisungsoperator einführen, indem wir ihn zunächst mit einer einfachen Zuweisungsanweisung vergleichen. Eine einfache Zuweisungsanweisung kann auch durch einen Zuweisungsausdruck ersetzt werden, obwohl er umständlich aussieht und definitiv in diesem Fall nicht der beabsichtigte Anwendungsfall ist.

```
x = 5
# wir können dies auch so schreiben:
(x := 5)  # möglich aber nicht empfohlen
# Die Klammern sind unbedingt notwendig
```

```
5
```

Zunächst mag man sich über die Klammern wundern, wenn man sich den Zuweisungsausdruck ansieht. Er ist nicht als Ersatz für die einfache “Zuweisungsanweisung” gedacht. Seine Hauptrolle besteht darin, innerhalb von Ausdrücken verwendet zu werden.

Der folgende Code zeigt einen geeigneten Anwendungsfall:

```
x = 4.56
z = (square := x**2) - (6.6 / square)
print(z)
```

```
20.476194644506
```

Obwohl man argumentieren könnte, dass der folgende Code möglicherweise noch klarer ist:

```
x = 4.56
square = x**2
z = square - (6.6 / square)
```

```
print(z)
```

```
20.476194644506
```

Hier ist ein weiteres Python-Codebeispiel, das ein ähnliches Szenario demonstriert. Zuerst mit dem Walross-Operator:

```
# Mit dem Walross-Operator

n = int(input("Bitte gebe eine Zahl ein: "))
if (square := n ** 2) > 3:
    print("Die Zahl ist größer als 3.")
    print(f"Das Quadrat von {n} ist {square}.")
```

```
Bitte geben Sie eine Zahl ein:  4

Die Zahl ist größer als 3.
Das Quadrat von 4 ist 16.
```

Nun ohne den Walross-Operator:

```
n = int(input("Bitte gebe eine Zahl ein: "))
square= n ** 2
if square > 3:
    print("Die Zahl ist größer als 3.")
    print(f"Das Quadrat von {n} ist {square}.")
```

```
Please give a number:  42

The number is greater than 3.
The square of 42 is 1764.
```

Vorteilhafte Anwendungen

Wir wollen nun einige Beispiele erkunden, in denen die Verwendung des Zuweisungsausdrucks wirklich vorteilhaft ist im Vergleich zu Code, der ihn nicht verwendet.

```
def f(x):
    return x + 4

numbers = [3, 7, 2, 9, 12]

odd_numbers = [result for x in numbers if (result := f(x)) % 2]
odd_numbers
```

```
[7, 11, 13]
```

Man kann beobachten, dass, wenn man sich entscheidet, den Zuweisungsausdruck in der Listenabstraktion zu vermeiden, man die Funktion zweimal aufrufen müsste. Daher macht die Verwendung des Zuweisungsausdrucks den Code in diesem Kontext effizienter.

```
def f(x):
    return x + 4
```

```
numbers = [3, 7, 2, 9, 12]

odd_numbers = [f(x) for x in numbers if  f(x) % 2]
odd_numbers
```

```
[7, 11, 13]
```

Anwendungsfall: Reguläre Ausdrücke Es gibt auch einen großen Vorteil, wenn man reguläre Ausdrücke verwendet. Wie in unseren vorherigen Beispielen präsentieren wir erneut zwei ähnliche Code-Snippets, das erste ohne und das zweite mit dem Walross-Operator:

```
import re

txt = """Der Python-Kurs begann am 2022-02-4
der andere Kurs am 2022-01-24
nur ein Datum pro Zeile, wenn überhaupt
die Daten können auch in diesem Format vorliegen 2020/10/15
oder 20-10-04"""

for line in txt.split('\n'):
    date = re.search(r'(\d{2,4})[-/](\d{2})[-/](\d{2})', line)
    if date:
        year, month, day = date.groups()
        print(f"Found date: {year}-{month}-{day}")
```

```
Found date: 2022-01-24
Found date: 2020-10-15
Found date: 20-10-04
```

Now, the code using assignment expressions:

```
import re

txt = """Der Python-Kurs begann am 2022-02-4
der andere Kurs am 2022-01-24
nur ein Datum pro Zeile, wenn überhaupt
die Daten können auch in diesem Format vorliegen 2020/10/15
oder 20-10-04"""


# Split the text into lines and search for dates in each line
for line in txt.split('\n'):
    # Using the assignment expression to find and store the date match
    if (date := re.search(r'(\d{2,4})[-/](\d{2})[-/](\d{2})', line)):
        year, month, day = date.groups()
        print(f"Found date: {year}-{month}-{day}")
```

```
Found date: 2022-01-24
Found date: 2020-10-15
Found date: 20-10-04
```

Zuweisungsausdruck beim Dateilesen

Dateien mit fester Breite Dateien, in denen jede Zeile die gleiche Länge hat, werden oft als "Dateien mit fester Breite" bezeichnet. In diesen Dateien ist jede Zeile (oder Datensatz) strukturiert, so dass sie eine vordefinierte Anzahl von Zeichen enthält, und die Felder innerhalb der Zeilen haben feste Positionen. Die Dateien können mit der read-Methode und dem Walross-Operator gelesen werden:

```
with open('python-kurse-info.txt') as fh:
    while ((data := fh.read(53)) != ''):
        print(data.rstrip())
```

```
Python Python-Kurs, Basics             Berlin     08
Python Kurs für Fortgeschrittene       Hamburg    06
Python OOP-Kurs                        Frankfurt  08
Python Python-Kurs, Basics             Freiburg   08
Python Kurs für Fortgeschrittene       München    97
Python OOP-Kurs                        Freiburg   12
Python Maschinelles Lernen             Stuttgart  06
Python OOP-Kurs                               Zürich     08
Python Python-Kurs, Basics             Basel      08
```

Jetzt dasselbe im traditionellen Codierungsstil:

```
with open('python-kurse-info.txt') as fh:
    data = fh.read(53)
    while data:
        print(data.rstrip())
        data = fh.read(53)
```

```
Python Python-Kurs, Basics             Berlin     08
Python Kurs für Fortgeschrittene       Hamburg    06
Python OOP-Kurs                        Frankfurt  08
Python Python-Kurs, Basics             Freiburg   08
Python Kurs für Fortgeschrittene       München    97
Python OOP-Kurs                        Freiburg   12
Python Maschinelles Lernen             Stuttgart  06
Python OOP-Kurs                               Zürich     08
Python Python-Kurs, Basics             Basel      08
```

Die Vorteile für das neueste Code-Snippet sind:

- Es verwendet einen traditionelleren Codierungsstil, der für einige Entwickler möglicherweise vertrauter ist.
- Es weist die Variable `data` explizit zu und verwendet sie, was den Code für diejenigen, die nicht mit Zuweisungsausdrücken vertraut sind, einfacher verständlich macht.

Ein bedeutender Nachteil dieses Ansatzes ist jedoch, dass er die explizite Zuweisung und Neuzuweisung der Variable `data` erfordert, was als weniger prägnant und etwas weniger elegant angesehen werden kann.

Verwendung von `readline` Die meisten Menschen verwenden eine `for`-Schleife, um über eine Textdatei zeilenweise zu iterieren. Mit dem Walross-Operator können wir jedoch auch elegant durch einen Text mit der Methode `readline` gehen:

```
word_to_find = "Maschinelles"
```

```
with open('python-kurse-info.txt') as file:
    while (line := file.readline()):
        if word_to_find in line:
            print(line)
```

```
Python Maschinelles Lernen             Stuttgart  06
```

Code snippet using `readline` but not the assignment expression:

```
word_to_find = "Maschinelles"
with open('python-kurse-info.txt') as file:
    line = "darf nur nicht leer sein, deshalb dieser Text"
    while line:
        line = file.readline()
        if word_to_find in line:
            print(line)
```

```
Python Maschinelles Lernen             Stuttgart  06
```

Nun mit dem Walross-Operator:

```
word_to_find = "OOP-Kurs"
with open('python-kurse-info.txt') as file:
    while (line := file.readline()):
        if word_to_find in line:
            print(line.rstrip())
```

```
Python OOP-Kurs                        Frankfurt  08
Python OOP-Kurs                        Freiburg   12
Python OOP-Kurs                               Zürich     08
```

Ein weiterer Anwendungsfall Im Kapitel über while-Schleifen unseres Python-Tutorials hatten wir ein kleines Zahlenratespiel:

```
import random

untere_grenze, obere_grenze = 1, 20
zu_ratende_zahl = random.randint(untere_grenze, obere_grenze)
vermutung = 0
while vermutung != zu_ratende_zahl:
    vermutung = int(input("Neue Zahl: "))
    if vermutung > zu_ratende_zahl:
        print("Zahl zu groß")
    elif vermutung < zu_ratende_zahl:
        print("Zahl zu klein")
else:
    print("Herzlichen Glückwunsch. Du hast es geschafft!")
```

```
Neue Zahl:  10

Zahl zu groß

Neue Zahl:  5

Zahl zu groß
```

```
Neue Zahl:  3

Zahl zu klein

Neue Zahl:  4

Herzlichen Glückwunsch. Du hast es geschafft!
```

Wie Sie sehen können, mussten wir `vermutung` auf null initialisieren, um die Schleife betreten zu können. Wir können die Initialisierung direkt in der Schleifenbedingung mit einem Zuweisungsausdruck durchführen und den gesamten Code dadurch vereinfachen:

```
import random

untere_grenze, obere_grenze = 1, 20
zu_ratende_zahl = random.randint(untere_grenze, obere_grenze)

while (vermutung := int(input("Neue Zahl: "))) != zu_ratende_zahl:
    if vermutung > zu_ratende_zahl:
        print("Zahl zu groß")
    elif vermutung < zu_ratende_zahl:
        print("Zahl zu klein")
else:
    print("Herzlichen Glückwunsch. Du hast es geschafft!")
```

```
Neue Zahl:  19

Zahl zu groß

Neue Zahl:  10

Zahl zu klein

Neue Zahl:  15

Zahl zu klein

Neue Zahl:  17

Herzlichen Glückwunsch. Du hast es geschafft!
```

Kritik an dem Walrus-Operator in Python

Wir haben zu Beginn dieser Seite erwähnt, dass einige Python-Programmierer schon lange auf dieses Konstrukt gewartet haben. Ein Grund, warum es nicht früher eingeführt wurde, war die Tatsache, dass es auch verwendet werden kann, um weniger lesbaren Code zu schreiben. Tatsächlich widerspricht die Anwendung des Walross-Operators mehreren Prinzipien, die im Zen von Python hervorgehoben werden. Um diese Verstöße zu erfassen, wollen wir uns den folgenden Szenarien zuwenden.

Im Zen von Python heißt es: “Explizit ist besser als implizit”. Der Walross-Operator verstößt gegen diese Anforderung des Zen von Python. Dies bedeutet, dass explizite Operationen immer besser sind als implizite Operationen. Der Walross-Operator weist einer Variablen einen Wert zu und gibt den Wert implizit zurück. Daher steht er im Widerspruch zum Konzept von “Explizit ist besser als implizit”.

Der folgende Python-Code-Schnipsel zeigt ein extremes Beispiel, das nicht empfohlen wird:

```
a, b, c = 1, 2, 3
x = 4
y = (c := (a := x*2.3) + (b := x*4.5 -3))
```

Was denken Sie über den folgenden Code, der den Python-Zuweisungsoperator innerhalb eines print-Funktionsaufrufs verwendet?

```
print((a := x*2.3) + (b := x*4.5 -3) + (x := 4))
```

```
28.2
```

Diese Python-Code-Schnipsel verstoßen sicherlich gegen zwei weitere Forderungen des Zen von Python:

1) Schönheit ist besser als Hässlichkeit (im Original: Beautiful is Better Than Ugly)
2) Komplexität ist besser als Kompliziertheit (original: Complex is Better Than Complicated)

Das Prinzip "Es sollte eine und vorzugsweise nur eine offensichtliche Art geben, es zu tun"3 aus dem Zen von Python betont die Bedeutung eines klaren und eindeutigen Ansatzes. Wenn wir die Möglichkeit haben, Zuweisungs- und Rückgabeoperationen in separate Anweisungen aufzuteilen, widerspricht die Wahl des weniger lesbaren und komplexeren Walross-Operators diesem Prinzip.

```
z = (x:=3) + (y:=4)
```

This is the preferred way to do itDies ist definitiv besser:

```
x, y = 3, 4
z = x + y
```

Footnotes

1 PEP 572

2 The purpose of this feature is not a new way to assign objects to variables, but it gives programmers a convenient way to assign variables in the middle of expressions. You might still be curious about the origin of the term "walrus operator." It's affectionately named this way because, with a touch of imagination, the operator's characters resemble the eyes and tusks of a walrus.

3 im Original: "There should be one and preferably only one obvious way to do it"

21 Python3 Print

21.0.1 print

Einführung

Jedes Computer-Programm muss prinzipiell mit seiner "Umgebung" oder "Außenwelt" korrespondieren. Dazu bietet fast jede Programmiersprache spezielle Ein-/Ausgabe-Funktionalitäten. Damit wird eine Interaktion/Kommunikation eines Programmes mit anderen Komponenten z.B. mit einer Datenbank oder seinen Benutzern ermöglicht. Eingaben kommen, - wie wir bereits an anderen Stellen in unserem Tutorial gesehen haben, - sehr häufig über die Tastatur und der entsprechende Python Befehl oder besser die entsprechende Python-Funktion zum Lesen von der Standardeingabe lautet input(). Wir haben in unseren Beispielen auch bereits gesehen, dass wir mittels print() in die Standardausgabe schreiben können. In diesem Kapitel unseres Tutorials wollen wir uns nun die print-Funktion im Detail ansehen. Weil es viele überlesen, wollen wir nochmals betonen, dass wir eben von einer print-Funktion und nicht von einer print-Anweisung gesprochen hatten. Wie wichtig dieser Unterschied ist, sieht man, wenn man sich ein beliebiges Python2-Programm nimmt und dieses unter Python3 laufen lässt bzw. versucht es laufen zu lassen. Dies funktioniert in den meisten Fällen nicht, und wir erhalten viele Fehlermeldungen. Eine sehr häufig auftretende Fehlermeldung gibt es in Zusammenhang mit den Ausgaben mit print, da die meisten Programme prints enthalten. Diesen typischen Fehler können wir auch in der interaktiven Python-Shell generieren:

```
print 42
```

```
  File "<ipython-input-1-6b6f29d376df>", line 1
    print 42
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print(42)?
```

Eine uns allen wohl bekannte Fehlermeldung: Wir haben die Klammern vergessen. "print" ist in Python3 eine Funktion und keine Anweisung mehr, wie dies vorher der Fall war, und deshalb müssen die Argumente, wie bei jeder anderen Funktion auch, innerhalb von einem Klammernpaar stehen. Wir können also obigen Fehler leicht durch Hinzufügen von Klammern beheben:

```
print(42)
```

```
42
```

Dies ist aber nicht der einzige Unterschied zum alten print. Auch das Ausgabeverhalten hat sich verändert. ### print-Funktion in Python3 Das Aufrufverhalten der print-Funktion ist wie folgt in Python3 definiert:

Die print-Funktion druckt beliebig viele Werte ("value1, value2, ...") aus, die durch Komma getrennt sind. Bei der Ausgabe werden die Werte standardmäßig durch Leerzeichen getrennt.

Im folgenden Beispiel sehen wir zwei print-Aufrufe, die jeweils zwei Werte, d.h. einen String und eine Float-Zahl ausgeben:

```
a= 3.564
print("a = ", a)
```

```
a =  3.564
```

```
print("a = \n", a)
```

```
a =
 3.564
```

Wir sehen im zweiten print des vorigen Beispiels, dass das Leerzeichen zwischen zwei Werten, also in unserem Fall die Werte "a = \textbackslash n" und "3.564", immer durch ein Leerzeichen getrennt werden, auch wenn die Ausgabe in einer neuen Zeile weitergeht. In Python 2 ist dies nicht so. Dort wird kein Leerzeichen in einer neuen Zeile ausgegeben. Leerzeichen werden dort nur zwischen zwei Werten ausgegeben, wenn kein Zeilenvorschub stattfindet. Mit dem Schlüsselwortparameter "sep" kann man den Separator, der zwischen den Werten ausgegeben wird, auf einen beliebigen Stringwert setzen, also auch zum Beispiel auf einen leeren String oder einen Smiley:

```
print("a","b")
```

```
a b
```

```
print("a","b",sep="")
```

```
ab
```

```
print(192,168,178,42,sep=".")
```

```
192.168.178.42
```

```
print("a","b",sep=":-)")
```

```
a:-)b
```

Nach der Ausgabe der Werte beendet die print-Funktion die Ausgabe mit einem Newline, wie wir im folgenden sehen:

```
for i in range(4):
    print(i)
```

```
0
1
2
3
```

Man kann dem Schlüsselwortparameter "end" einen beliebigen String zuweisen, der dann statt dem Default-Wert "\textbackslash n" verwendet wird. Das erlaubt uns z.B. mehrere Werte in einer Zeile auszugeben:

```
for i in range(4):
    print(i, end=" ")
```

```
0 1 2 3
```

```
for i in range(4):
    print(i, end=" :-) ")
```

```
0 :-) 1 :-) 2 :-) 3 :-)
```

Die Ausgabe der print-Funktion wird in einen Datenstrom (engl. stream) geleitet. Standardmäßig wird dazu die Standardausgabe benutzt, also “sys.stdout”. Mit dem Schlüsselwort “file” sind wir schließlich in der Lage die Ausgabe in eine Datei umzuleiten.

```
fh = open("daten.txt","w")
print("42 ist die Antwort, aber was ist die Frage?", file=fh)
fh.close()
```

Wir können feststellen, dass print nun keine Ausgabe mehr in der interaktiven Shell erzeugt. Die Ausgabe erfolgt nun in die Datei “daten.txt”. Auf diese Art ist es nicht nur möglich Ausgaben in eine beliebige Datei umzulenken, sondern auch in den Standardfehlerkanal:

```
import sys
# Ausgabe in Standardfehlerkanal:
print("Error: 42", file=sys.stderr)
```

```
Error: 42
```

22 Python3 Formatierte Ausgabe

22.0.1 Formatierte Ausgabe

Wege die Ausgabe zu formatieren

In diesem Kapitel unseres Python-Tutorials werden wir uns intensiv mit den verschiedenen Methoden beschäftigen, mit denen man die Ausgaben formatieren bzw. formatierte Strings erzeugen kann. Wir stellen die verschiedenen Arten vor, aber wir empfehlen die format-Methode der Stringklasse zu benutzen, die sich kurz vor Ende des Kapitels befindet. Die format-Methode ist die bei weitem flexibelste.

Bisher hatten wir die print-Funktion auf zwei Arten benutzt, wenn wir beispielsweise zwei Werte ausgeben wollten:

Die einfachste Art, aber sicherlich nicht die eleganteste: Wir benutzen print mit einer kommaseparierten Liste von Werten, um die Ergebnisse auszugeben, wie wir im folgenden Beispiel sehen können. Alle Werte werden durch Leerzeichen getrennt, was dem Default-Verhalten entspricht. Wir können die Default-Trennung durch einen beliebigen String ersetzen. Dazu müssen wir diesen String lediglich durch den Schlüsselwort-Parameter "sep" der Print-Funktion ersetzen:

```
q = 459
p = 0.098
print(q, p, p * q)
```

```
459 0.098 44.982
```

```
print(q, p, p * q, sep=",")
```

```
459,0.098,44.982
```

```
print(q, p, p * q, sep=" :-) ")
```

```
459 :-) 0.098 :-) 44.982
```

Alternativ können wir auch aus den Werten mittels der String-Konkatenation einen neuen String konstruieren:

```
print(str(q) + " " + str(p) + " " + str(p * q))
```

```
459 0.098 44.982
```

Die zweite Methode ist schlechter, - da komplizierter, - als die erste.

Die alte Methode im Stil von printf und sprintf

Statt “alte” Methode sollten wir besser “veraltete” Methode sagen, da dieses Vorgehen keinen guten Python-Stil mehr darstellt.

Gibt es ein printf in Python? Das ist die brennende Frage für Python-Anfänger, die von C, Perl, Bash oder anderen Programmiersprachen kommen, die dieses Statement bzw. diese Funktion benutzen. Zu sagen, dass Python eine print-Funktion und keine printf-Funktion hat ist nur die eine Seite der Medaille. Also gibt es doch eine “printf”-Funktion in Python? Nein, aber die Funktionalität des “alten” printf wurde in anderer Form übernommen. Zu diesem Zweck wird der Modulo-Operator “%” von der String-Klasse überladen. Deshalb spricht man bei dem überladenen Operator “%” der String-Klasse auch häufig vom String-Modulo-Operator, obwohl es eigentlich wenig mit der eigentlich Modulo-Arithmetik zu tun hat. Manchmal wird er auch synonym als String-Modulus-Operator genannt. Ein weiterer Begriff, für diese Operation lautet “String-Interpolation”, weil bei dieser Operation Werte, wie Integers, Floats usw., in den zu erzeugenden formatierten String eingefügt also interpoliert werden. Den mittels der String-Interpolation erzeugten formatierten String, kann man dann z.B. mittels print (also nicht printf) ausgeben. Aber man kann diesen formatierten String auch auf andere Art im Programm weiter verwenden, wie ihn zum Beispiel in eine Datenbank übertragen. Seit Python 2.6 eingeführt worden ist, wird jedoch von der Python-Gemeinde empfohlen die String-Methode “format” stattdessen zu verwenden. Unglücklicherweise konnten man sich auch bei dem Design von Python3 nicht von der veralteten String-Interpolations-Methode trennen. Außerdem wird die veraltete Methode noch allzu häufig benutzt. Aus diesem Grunde gehen wir auch in unserem Tutorial ausgiebig darauf ein. Um fremden Python-Code verstehen zu können, der die String-Interpolations-Methode benutzt, ist es nötig den Mechanismus zu verstehen. Wahrscheinlich - oder hoffentlich - wird man sich jedoch bei der Weiterentwicklung von Python eines Tages von dieser Methode trennen.

Das folgende Diagramm verbildlicht die Arbeitsweise des String-Modulo-Operators:

Auf der linken Seite des String-Modulo-Operators befindet sich der sogenannte Format-String und auf der rechten Seite steht ein Tupel mit den Werten, die in den Format-String interpoliert bzw. eingefügt werden sollen. Die Werte können Literale, Variablen oder beliebige arithmetische Ausdrücke sein.

Der Formatstring enthält Platzhalter. In unserem Beispiel befinden sich zwei davon: “%5d” und “%8.2f”.

Die allgemeine Syntax für diese Platzhalter lautet:

Schauen wir uns einmal die Platzhalter unseres Beispieles genauer an: Der zweite, also “%8.2f”, ist eine Formatbeschreibung für eine Floatzahl. Wie auch alle anderen Platzhalter wird er durch ein “%”-Zeichen eingeleitet. Dieses wird von einer Zahl gefolgt, die die totale Anzahl der Zeichen angibt, mit der die Zahl ausgegeben werden soll. Diese Anzahl beinhaltet auch den Dezimalpunkt und alle Ziffern vor und nach dem Punkt. Unsere Floatzahl 59.058 muss also mit 8 Zeichen dargestellt werden. Der Dezimalanteil der Zahl ist auf 2 gesetzt, dies ist die Zahl hinter dem “.” beim Platzhalter. Das bedeutet, dass die Anzahl der Nachkommastellen auf 2 gesetzt ist. Das letzte Zeichen im Platzhalt ist der Buchstabe “f”. Wie sich leicht vermuten lässt, steht er für “float”.

Schaut man sich die Ausgabe an, erkennt man, dass die Zahl “59.058” als ” 59.06” ausgegeben wird. Man sieht auch, dass die Ausgabe nicht etwa nach zwei Stellen abgebrochen wird, sondern dass eine Rundung erfolgt. Außerdem werden von links drei Leerzeichen an die Zahl bei der Ausgabe angefügt.

Der erste Platzhalter "%5d" dient als Beschreibung für die erste Komponente unseres Tupels, d.h. die Ganzzahl (integer) 453. Diese Zahl soll mit 5 Zeichen ausgedruckt werden. Da 453 aber nur aus 3 Stellen besteht, werden der Zahl in der Ausgabe zwei Leerzeichen vorangestellt. Viele werden statt "%5d" einen Platzhalter der Art "%5i" erwartet haben, da es sich ja um eine Integer-Zahl handelt. Worin besteht der Unterschied? Ganz einfach: Es gibt keinen Unterschied zwischen "d" und "i". Beide werden zur Formatierung von Integers genutzt.

Die Vorteile bzw. die Schönheit der formatierten Ausgabe kann man nur ermessen, wenn man mehrere gleichermaßen formatierte Ausgaben ausgibt. Im folgenden Beispiel zeigen wir, wie es aussieht, wenn wir 5 Fließzahlen (floats) gleich formatiert mit dem Platzhalter "%6.2f" in verschiedenen Zeilen ausgeben:

Platzhaltersymbol

Bedeutung

d

```
    <td>Vorzeichenbehaftete Ganzzahl (Integer, dezimal).</td>
    </tr>
<tr><td valign="baseline"><tt>i</tt></td>
    <td>Vorzeichenbehaftete Ganzzahl (Integer, dezimal).</td>
    </tr>
<tr><td valign="baseline"><tt>o</tt></td>
    <td>vorzeichenlose Ganzzahlen (oktal)</td>
    </tr>
<tr><td valign="baseline"><tt>u</tt></td>
    <td>vorzeichenlose Ganzzahlen (dezimal)</td>
    </tr>
<tr><td valign="baseline"><tt>x</tt></td>
    <td>vorzeichenlose Ganzzahlen (hexadezimal)</td>
    </tr>
<tr><td valign="baseline"><tt>X</tt></td>
    <td>vorzeichenlose Ganzzahlen (hexadezimal), Uppercase</td>
    </tr>
<tr><td valign="baseline"><tt>e</tt></td>
    <td>Fließkommazahl im Exponentialformat (in Kleinbuchstaben).</td>
    </tr>
<tr><td valign="baseline"><tt>E</tt></td>
    <td>Fließkommazahl im Exponentialformat (in Großbuchstaben).</td>
    </tr>
<tr><td valign="baseline"><tt>f</tt></td>
    <td>wie e</td>
    </tr>
<tr><td valign="baseline"><tt>F</tt></td>
    <td>wie E</td>
    </tr>
<tr><td valign="baseline"><tt>g</tt></td>
    <td>Wie "<tt>e</tt>", wenn der Exponent größer ist als -4 oder
          oder kleiner als die Präzision. Ansonsten wie "<tt>f</tt>"</td>
    </tr>
<tr><td valign="baseline"><tt>G</tt></td>
```

```
    <td>wie "<tt>E</tt>" und analog zu "<tt>g</tt>"</td>
    </tr>
<tr><td valign="baseline"><tt>c</tt></td>
    <td>ein Zeichen</td>
    </tr>
<tr><td valign="baseline"><tt>s</tt></td>
    <td>Eine Zeichenkette (String); beliebige Python-Objekte werden in String mittels de
    </tr>
<tr><td valign="baseline"><tt>r</tt></td>
    <td>siehe <tt>s</tt></td>
    </tr>
<tr><td valign="baseline"><tt>%</tt></td>
    <td>Es findet keine Argument-Konvertierung statt, es wird ein "<tt>%</tt>"-
Zeichen ausgegeben.</td>
    </tbody>
```

Die folgenden Beispiele zeigen einige exemplarische Anwendungen der Konvertierungsmöglichkeiten der vorigen Tabelle:

```
print("%10.3e"% (356.08977))
```

```
 3.561e+02
```

```
print("%10.3E"% (356.08977))
```

```
 3.561E+02
```

```
print("%10o"% (25))
```

```
        31
```

```
print("%10.3o"% (25))
```

```
       031
```

```
print("%10.5o"% (25))
```

```
     00031
```

```
print("%5x"% (47))
```

```
   2f
```

```
print("%5.4x"% (47))
```

```
 002f
```

```
print("%5.4X"% (47))
```

```
 002F
```

```
print("Ein Prozentzeichen: %% " % ())
```

```
Ein Prozentzeichen: %
```

Flag

Bedeutung

#

```
    <td>Wird dieses Zeichen mit o, x oder X benutzt, wird der jeweilige Wert mit dem entspr
```

0

```
    <td>Das Ergebnis der Umwandlung wird mit Nullen aufgefüllt.</td></tr>
<tr><td valign="baseline"><tt>-</tt></td>
    <td>Das Ergebnis der Umwandlung wird linksbündig ausgegeben.</td></tr>
<tr><td valign="baseline"><tt> </tt></td>
    <td>Falls kein Vorzeichen (beispielsweise ein Minuszeichen) ausgegeben wird, wird ein L
<tr><td valign="baseline"><tt>+</tt></td>
    <td>Das Ergebnis der Umwandlung wird mit einem Vorzeichen versehen ("<tt>+</tt>" oder "
Flag).</td></tr></tbody>
```

Beispiele:

```
print("%#5X"% (47))
```

```
 0X2F
```

```
print("%5X"% (47))
```

```
   2F
```

```
print("%#5.4X"% (47))
```

```
0X002F
```

```
print("%#5o"% (25))
```

```
 0o31
```

```
print("%+d"% (42))
```

```
+42
```

```
print("% d"% (42))
```

```
 42
```

```
print("%+2d"% (42))
```

```
+42
```

```
print("% 2d"% (42))
```

```
 42
```

```
print("%2d"% (42))
```

```
42
```

Obwohl es so ausschauen mag, ist die Formatierung nicht Teil der Print-Funktion. Schaut man sich die vorigen Beispiele genauer an, sieht man, dass wir einen formatierten String an die Print-Funktion übergeben haben. Oder anders ausgedrückt: Wenn die String-Interpolation auf einen String angewendet wird, liefert sie einen String zurück. Dieser String wird dann an print übergeben. Dies bedeutet wiederum, dass wir die String-Interpolation auch hätten "zweistufig" anwenden können, d.h. wir hätten erst einen formatierten String erzeugt, diesen einer Variablen zugewiesen und diese Variable dann an print übergeben:

```
s = "Preis: $ %8.2f"% (356.08977)
print(s)
```

```
Preis: $   356.09
```

Der pythonische Weg: Die String-Methode "format"

Was die String-Methode format betrifft, ist die help-Funktion von Python nicht sehr hilfreich. Alles was sie uns sagt, ist dieses:

| format(…) | S.format(*args, **kwargs) -> str |
| Return a formatted version of S, using substitutions from args and kwargs. | The substitutions are identified by braces ('{' and '}'). |

Die String-Methode "format" wurde mit Python 2.6 als Ersatz für die String-Interpolation eingeführt.

Ganz allgemein sieht "format" wie folgt aus:

Der Format-String ist ein String, der ein oder mehrere Format-Codes innerhalb eines konstanten Textes enthält, d.h. Felder, die ersetzt werden sollen. Die zu "ersetzenden Felder" sind von geschweiften Klammern umgeben. Eine geschweifte Klammer und der von ihr umschlossene "Code" wird mit dem formatierten Wert aus dem korrespondierenden Argument ersetzt. Nach welchen Regeln dies zu erfolgen hat, werden wir im folgenden erläutern. Alles andere, was nicht von geschweiften Klammern umschlossen ist, wird wörtlich, also ohne Änderungen, ausgedruckt. Möchte man eine öffnende oder eine schließende geschweifte Klammer ausgeben, muss man diese verdoppeln, also "{{" oder "}}".

Es gibt zwei Arten von Argumenten für die format-Methode. Die Argumentenliste startet mit 0 oder mehr Positionsargumenten (p0, p1, …), die von 0 oder mehr Schlüsselwortargumenten der Form name=value gefolgt werden können.

Ein Positionsparameter der format-Methode kann benutzt werden, indem man den Index des Parameters nach öffnenden geschweiften Klammer angibt, d.h. {0} für den ersten Parameter, {1} für den zweiten und so weiter. Der Index des Positionsparameter nach der öffnenden geschweiften Klammer kann von einem Doppelpunkt und einem Formatstring gefolgt werden, der dem des String-Interpolationsparameters ähnelt, den wir zu Beginn des Kapitels besprochen hatten, also zum Beispiel {0:5d} Falls die Positionsparameter in der Reihenfolge benutzt werden, in der sie in der Parameterliste stehen, kann die Angabe des Indexes innerhalb der geschweiften Klammern entfallen, d.h. '{} {} {}' entspricht '{0} {1} {2}'. Um es nochmals klar zu stellen: Will man sie in einer anderen Reihenfolge benutzen, sind die Angaben der Indexpositionen dringend notwendig, wie z.B. hier '{2} {1} {0}'

Im folgenden Diagramm zeigen wir anhand eines Beispieles wie die Stringmethode "format" für zwei Parameter funktioniert:

Beispiele mit Positionsparametern:

```
"Erstes Argument: {0}, zweites: {1}".format(47,11)
```

```
"Zweites Argument: {1}, erstes: {0}".format(47,11)
```

```
"Zweites Argument: {1:3d}, erstes: {0:7.2f}".format(47.42,11)
```

```
"Erstes Argument: {}, zweites: {}".format(47,11)

# Argumente können auch mehrmals verwendet werden:
```

```
"precisions: {0:6.2f} or {0:6.3f}".format(1.4148)
```

Im folgenden Beispiel demonstrieren wir, wie Schlüsselwortparameter mit der format-Methode benutzt werden können:

```
"Artikel: {a:5d},  Preis: {p:8.2f}".format(a=453, p=59.058)
```

```
'Artikel:   453,  Preis:    59.06'
```

Mit der format-Methode ist es auch möglich Daten links- und rechtsbündig auszugeben. Dazu müssen wir der Formatierungsanweisung ein "<" (linksbündig) oder ein ">" (rechtsbündig) voranstellen. Wir demonstrieren dies an den folgenden Beispielausgaben:

```
"{0:<20s} {1:6.2f}".format('Spam & Eggs:', 6.99)
```

```
'Spam & Eggs:           6.99'
```

```
"{0:>20s} {1:6.2f}".format('Spam & Ham:', 7.99)
```

```
'         Spam & Ham:   7.99'
```

```
"{0:<20} {1:6.2f}".format('Spam ', 0)
```

```
'Spam                   0.00'
```

If the width field is preceded by a zero (‘0’) character,
sign-aware zero-padding for numeric types will be enabled.

```
x = 378
print("The value is {0:06d}".format(x))
```

```
The value is 000378
```

```
x = -378
print("The value is {0:06d}".format(x))
```

```
The value is -00378
```

```
"{0:>20} {1:6.2f}".format('Spam & Ham:', 7.99)
```

```
'          Spam & Ham:   7.99'
```

Ausrichtungs-Option

Bedeutung

‘<’

Das Feld wird linksbündig innerhalb des vorhandenen Raumes ausgegeben. Strings werden standardmäßig linksbündig ausgegeben.

‘>’

Das Feld wird rechtsbündig innerhalb des vorhandenen Raumes ausgegeben. Numerische Werte werden standardmäßig rechtsbündig ausgegeben.

‘0’

Wenn man das Formatfeld mit einer führenden Null (‘0’) versieht, erfolgt ein Auffüllen mit Nullen unter Beachtung eines möglichen Vorzeichens.

‘,’

Mit dieser Option erhält man Tausender Gruppierungen, die mit Komma getrennt sind:

‘=’

Die Auffüllzeichen (padding characters) werden zwischen Vorzeichen, falls ein Vorzeichen ausgegeben wird und dem eigentlichen Beginn der Ziffern einer Zahl eingeführt. Damit können Felder der Art “+000000120” ausgegeben werden. Diese Ausrichtungsoption kann nur auf numerische Werte angewendet werden.

‘^’

Ein Feld wird zentriert innerhalb des zur Verfügung stehenden Raumes ausgegeben.

Wenn keine minimale Feldlänge angegeben wird, entspricht die Feldlänge immer der Länge der Daten, die ein Feld enthält. In diesen Fällen haben die Ausrichtungsoptionen keine Bedeutung.

Zusätzlich können wir noch die Ausgabe mittels der sign-Option beeinflussen. Diese Optionen sind nur für numerische Werte gültig:

sign-Option

Bedeutung

‘+’

Es soll immer ein Vorzeichen ausgegeben werden, also unabhängig davon, ob es sich um eine positive oder negative Zahl handelt.

‘-’

Ein Vorzeichen soll nur bei negativen Zahlen verwendet werden.

space

Statt eines “+”-Zeichens wird bei positiven Zahlen ein Leerzeichen ” ” vorangestellt. Bei negativen Zahlen ein Minus-Zeichen “-”.

Benutzung von Dictionaries beim Aufruf der “format”-Methode

In den vorigen Kapitel haben wir gesehen, dass wir zwei Möglichkeiten haben Werte mit “format” zu formatieren:

- Benutzung des Index bzw. der Position des Argumentes:

```
print("Die Hauptstadt von {0:s} ist {1:s}".format("Ontario","Toronto"))
```

```
Die Hauptstadt von Ontario ist Toronto
```

Um es nochmals zu erwähnen: Im vorigen Beispiel hätten wir auch geschweifte Klammern mit leeren Inhalt benutzen können! - Benutzung von Schlüsselwort-Parametern:

```
print("Die Hauptstadt von {province} ist {capital}".format(province="Ontario",␣
↪  capital="Toronto"))
```

```
Die Hauptstadt von Ontario ist Toronto
```

Im zweiten Fall könnten wir auch ein Dictionary verwenden, wie wir im folgenden Code sehen können:

```
data = dict(province="Ontario",capital="Toronto")
data
```

```
{'province': 'Ontario', 'capital': 'Toronto'}
```

```
print("The capital of {province} is {capital}".format(**data))
```

```
The capital of Ontario is Toronto
```

Das doppelte Sternchen (“*“) vor data wandelt data automatisch in die Form ‘province=“Ontario”,capital=“Toronto” ’. Schauen wir uns das folgende Python-Programm an:

```
capital_country = {"Vereinigte Staaten" : "Washington",
                   "US" : "Washington",
                   "Kanada" : "Ottawa",
                   "Deutschland": "Berlin",
```

```
                    "Frankreich" : "Paris",
                    "England" : "London",
                    "Großbritannien" : "London",
                    "Schweiz" : "Bern",
                    "Österreich" : "Wien",
                    "Niederlande" : "Amsterdam"}

print("Länder und Ihre Hauptstädte:")
for c in capital_country:
    print("{country}: {capital}".format(country=c, capital=capital_country[c]))
```

```
Länder und Ihre Hauptstädte:
Vereinigte Staaten: Washington
US: Washington
Kanada: Ottawa
Deutschland: Berlin
Frankreich: Paris
England: London
Großbritannien: London
Schweiz: Bern
Österreich: Wien
Niederlande: Amsterdam
```

Das vorige Programm können wir so umschreiben, dass im Aufruf der format-Methode das Dictionary direkt verwendet wird.

```
capital_country = {"Vereinigte Staaten" : "Washington",
                    "US" : "Washington",
                    "Kanada" : "Ottawa",
                    "Deutschland": "Berlin",
                    "Frankreich" : "Paris",
                    "England" : "London",
                    "Großbritannien" : "London",
                    "Schweiz" : "Bern",
                    "Österreich" : "Wien",
                    "Niederlande" : "Amsterdam"}

print("Countries and their capitals:")
for c in capital_country:
    format_string = c + ": {" + c + "}"
    print(format_string.format(**capital_country))
```

```
Countries and their capitals:
Vereinigte Staaten: Washington
US: Washington
Kanada: Ottawa
Deutschland: Berlin
Frankreich: Paris
England: London
Großbritannien: London
Schweiz: Bern
Österreich: Wien
Niederlande: Amsterdam
```

Der Ausdruck ist gleich wie beim ersten Programm.

Benutzung von lokalen Variablen in “format”

“locals” ist eine Funktion, die ein Dictionary mit den lokal definierten Namen und ihren aktuellen Werten zurückliefert. Die Variablen sind die Schlüssel dieses Dictionary und die Werte zu den Schlüsseln entsprechen den Werten der Variablen:

```
a = 42
b = 47
def f(): return 42
locals()
```

```
{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive environment',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': <module 'builtins' (built-in)>,
 '_ih': ['',
  's = "Programming"\ns.rjust(15)',
  's.rjust(15, "~")',
  'account_number = "43447879"\naccount_number.zfill(12)',
  '# can be emulated with rjust:\n\naccount_number.rjust(12,"0")',
  's = "Training"\ns.ljust(12)',
  's.ljust(12,":")',
  's = "Python"\ns.center(10)   ',
  's.center(10,"*") ',
  'capital_country = {"Vereinigte Staaten" : "Washington", \n
  ↪  "US" : "Washington", \n                    "Kanada" : "Ottawa",\n
  ↪  "Deutschland": "Berlin",\n                    "Frankreich" : "Paris",\n
  ↪  "England" : "London",\n                    "Großbritannien" : "London",\n
  ↪  "Schweiz" : "Bern",\n                    "Österreich" : "Wien",\n
  ↪  "Niederlande" : "Amsterdam"}\n\nprint("Countries and their capitals:")\nfor
  ↪  c in capital_country:\n    format_string = c + ": {" + c + "}" \n
  ↪  print(format_string.format(**capital_country))',
  'a = 42\nb = 47\ndef f(): return 42',
  'locals()',
  'print("a={a}, b={b} and f={f}".format(**locals()))',
  'a = 42\nb = 47\ndef f(): return 42\nlocals()'],
 '_oh': {1: '    Programming',
  2: '~~~~Programming',
  3: '000043447879',
  4: '000043447879',
  5: 'Training    ',
  6: 'Training:::',
  7: '  Python  ',
  8: '**Python**',
  11: {...}},
 '_dh': ['C:\\Users\\melis\\Dropbox (Bodenseo)\\Bodenseo Team
 ↪  Folder\\melisa\\notebooks_de'],
 'In': ['',
  's = "Programming"\ns.rjust(15)',
  's.rjust(15, "~")',
  'account_number = "43447879"\naccount_number.zfill(12)',
  '# can be emulated with rjust:\n\naccount_number.rjust(12,"0")',
  's = "Training"\ns.ljust(12)',
  's.ljust(12,":")',
```

```
 's = "Python"\ns.center(10)   ',
 's.center(10,"*") ',
 'capital_country = {"Vereinigte Staaten" : "Washington", \n
 ↪  "US" : "Washington", \n                    "Kanada" : "Ottawa",\n
 ↪  "Deutschland": "Berlin",\n                    "Frankreich" : "Paris",\n
 ↪  "England" : "London",\n                    "Großbritannien" : "London",\n
 ↪  "Schweiz" : "Bern",\n                   "Österreich" : "Wien",\n
 ↪  "Niederlande" : "Amsterdam"}\n\nprint("Countries and their capitals:")\nfor
 ↪  c in capital_country:\n    format_string = c + ": {" + c + "}" \n
 ↪  print(format_string.format(**capital_country))',
 'a = 42\nb = 47\ndef f(): return 42',
 'locals()',
 'print("a={a}, b={b} and f={f}".format(**locals()))',
 'a = 42\nb = 47\ndef f(): return 42\nlocals()'],
'Out': {1: '    Programming',
 2: '~~~~Programming',
 3: '000043447879',
 4: '000043447879',
 5: 'Training    ',
 6: 'Training::::',
 7: '  Python  ',
 8: '**Python**',
 11: {...}},
'get_ipython': <bound method InteractiveShell.get_ipython of
↪  <ipykernel.zmqshell.ZMQInteractiveShell object at 0x000001BC787B7388>>,
'exit': <IPython.core.autocall.ZMQExitAutocall at 0x1bc787f4088>,
'quit': <IPython.core.autocall.ZMQExitAutocall at 0x1bc787f4088>,
'_': {...},
'__': '**Python**',
'___': '  Python  ',
'_i': 'print("a={a}, b={b} and f={f}".format(**locals()))',
'_ii': 'locals()',
'_iii': 'a = 42\nb = 47\ndef f(): return 42',
'_i1': 's = "Programming"\ns.rjust(15)',
's': 'Python',
'_1': '    Programming',
'_i2': 's.rjust(15, "~")',
'_2': '~~~~Programming',
'_i3': 'account_number = "43447879"\naccount_number.zfill(12)',
'account_number': '43447879',
'_3': '000043447879',
'_i4': '# can be emulated with rjust:\n\naccount_number.rjust(12,"0")',
'_4': '000043447879',
'_i5': 's = "Training"\ns.ljust(12)',
'_5': 'Training    ',
'_i6': 's.ljust(12,":")',
'_6': 'Training::::',
'_i7': 's = "Python"\ns.center(10)   ',
'_7': '  Python  ',
'_i8': 's.center(10,"*") ',
'_8': '**Python**',
```

```
 '_i9': 'capital_country = {"Vereinigte Staaten" : "Washington", \n
 ↪  "US" : "Washington", \n                    "Kanada" : "Ottawa",\n
 ↪  "Deutschland": "Berlin",\n                    "Frankreich" : "Paris",\n
 ↪  "England" : "London",\n                    "Großbritannien" : "London",\n
 ↪  "Schweiz" : "Bern",\n                    "Österreich" : "Wien",\n
 ↪  "Niederlande" : "Amsterdam"}\n\nprint("Countries and their capitals:")\nfor
 ↪  c in capital_country:\n    format_string = c + ": {" + c + "}" \n
 ↪  print(format_string.format(**capital_country))',
 'capital_country': {'Vereinigte Staaten': 'Washington',
  'US': 'Washington',
  'Kanada': 'Ottawa',
  'Deutschland': 'Berlin',
  'Frankreich': 'Paris',
  'England': 'London',
  'Großbritannien': 'London',
  'Schweiz': 'Bern',
  'Österreich': 'Wien',
  'Niederlande': 'Amsterdam'},
 'c': 'Niederlande',
 'format_string': 'Niederlande: {Niederlande}',
 '_i10': 'a = 42\nb = 47\ndef f(): return 42',
 'a': 42,
 'b': 47,
 'f': <function __main__.f()>,
 '_i11': 'locals()',
 '_11': {...},
 '_i12': 'print("a={a}, b={b} and f={f}".format(**locals()))',
 '_i13': 'a = 42\nb = 47\ndef f(): return 42\nlocals()'}
```

```

```

```
{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive environment',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': <module 'builtins' (built-in)>,
 '_ih': ['',
  's = "Programming"\ns.rjust(15)',
  's.rjust(15, "~")',
  'account_number = "43447879"\naccount_number.zfill(12)',
  '# can be emulated with rjust:\n\naccount_number.rjust(12,"0")',
  's = "Training"\ns.ljust(12)',
  's.ljust(12,":")',
  's = "Python"\ns.center(10)   ',
  's.center(10,"*") ',
  'capital_country = {"Vereinigte Staaten" : "Washington", \n
  ↪  "US" : "Washington", \n                    "Kanada" : "Ottawa",\n
  ↪  "Deutschland": "Berlin",\n                    "Frankreich" : "Paris",\n
  ↪  "England" : "London",\n                    "Großbritannien" : "London",\n
  ↪  "Schweiz" : "Bern",\n                    "Österreich" : "Wien",\n
  ↪  "Niederlande" : "Amsterdam"}\n\nprint("Countries and their capitals:")\nfor
  ↪  c in capital_country:\n    format_string = c + ": {" + c + "}" \n
  ↪  print(format_string.format(**capital_country))',
  'a = 42\nb = 47\ndef f(): return 42',
  'locals()'],
```

```
'_oh': {1: '    Programming',
 2: '~~~~Programming',
 3: '000043447879',
 4: '000043447879',
 5: 'Training    ',
 6: 'Training::::',
 7: '  Python  ',
 8: '**Python**'},
'_dh': ['C:\\Users\\melis\\Dropbox (Bodenseo)\\Bodenseo Team
↪  Folder\\melisa\\notebooks_de'],
'In': ['',
 's = "Programming"\ns.rjust(15)',
 's.rjust(15, "~")',
 'account_number = "43447879"\naccount_number.zfill(12)',
 '# can be emulated with rjust:\n\naccount_number.rjust(12,"0")',
 's = "Training"\ns.ljust(12)',
 's.ljust(12,":")',
 's = "Python"\ns.center(10)   ',
 's.center(10,"*") ',
 'capital_country = {"Vereinigte Staaten" : "Washington", \n
 ↪  "US" : "Washington", \n                    "Kanada" : "Ottawa",\n
 ↪  "Deutschland": "Berlin",\n                    "Frankreich" : "Paris",\n
 ↪  "England" : "London",\n                    "Großbritannien" : "London",\n
 ↪  "Schweiz" : "Bern",\n                   "Österreich" : "Wien",\n
 ↪  "Niederlande" : "Amsterdam"}\n\nprint("Countries and their capitals:")\nfor
 ↪  c in capital_country:\n    format_string = c + ": {" + c + "}" \n
 ↪  print(format_string.format(**capital_country))',
 'a = 42\nb = 47\ndef f(): return 42',
 'locals()'],
'Out': {1: '    Programming',
 2: '~~~~Programming',
 3: '000043447879',
 4: '000043447879',
 5: 'Training    ',
 6: 'Training::::',
 7: '  Python  ',
 8: '**Python**'},
'get_ipython': <bound method InteractiveShell.get_ipython of
↪  <ipykernel.zmqshell.ZMQInteractiveShell object at 0x000001BC787B7388>>,
'exit': <IPython.core.autocall.ZMQExitAutocall at 0x1bc787f4088>,
'quit': <IPython.core.autocall.ZMQExitAutocall at 0x1bc787f4088>,
'_': '**Python**',
'__': '  Python  ',
'___': 'Training::::',
'_i': 'a = 42\nb = 47\ndef f(): return 42',
'_ii': 'capital_country = {"Vereinigte Staaten" : "Washington", \n
↪  "US" : "Washington", \n                    "Kanada" : "Ottawa",\n
↪  "Deutschland": "Berlin",\n                    "Frankreich" : "Paris",\n
↪  "England" : "London",\n                    "Großbritannien" : "London",\n
↪  "Schweiz" : "Bern",\n                   "Österreich" : "Wien",\n
↪  "Niederlande" : "Amsterdam"}\n\nprint("Countries and their capitals:")\nfor
↪  c in capital_country:\n    format_string = c + ": {" + c + "}" \n
↪  print(format_string.format(**capital_country))',
'_iii': 's.center(10,"*") ',
'_i1': 's = "Programming"\ns.rjust(15)',
's': 'Python',
'_1': '    Programming',
'_i2': 's.rjust(15, "~")',
```

```
 '_2': '~~~~Programming',
 '_i3': 'account_number = "43447879"\naccount_number.zfill(12)',
 'account_number': '43447879',
 '_3': '000043447879',
 '_i4': '# can be emulated with rjust:\n\naccount_number.rjust(12,"0")',
 '_4': '000043447879',
 '_i5': 's = "Training"\ns.ljust(12)',
 '_5': 'Training    ',
 '_i6': 's.ljust(12,":")',
 '_6': 'Training::::',
 '_i7': 's = "Python"\ns.center(10)   ',
 '_7': '  Python  ',
 '_i8': 's.center(10,"*") ',
 '_8': '**Python**',
 '_i9': 'capital_country = {"Vereinigte Staaten" : "Washington", \n
 ↪  "US" : "Washington", \n                    "Kanada" : "Ottawa",\n
 ↪  "Deutschland": "Berlin",\n                    "Frankreich" : "Paris",\n
 ↪  "England" : "London",\n                    "Großbritannien" : "London",\n
 ↪  "Schweiz" : "Bern",\n                   "Österreich" : "Wien",\n
 ↪  "Niederlande" : "Amsterdam"}\n\nprint("Countries and their capitals:")\nfor
 ↪  c in capital_country:\n    format_string = c + ": {" + c + "}" \n
 ↪  print(format_string.format(**capital_country))',
 'capital_country': {'Vereinigte Staaten': 'Washington',
  'US': 'Washington',
  'Kanada': 'Ottawa',
  'Deutschland': 'Berlin',
  'Frankreich': 'Paris',
  'England': 'London',
  'Großbritannien': 'London',
  'Schweiz': 'Bern',
  'Österreich': 'Wien',
  'Niederlande': 'Amsterdam'},
 'c': 'Niederlande',
 'format_string': 'Niederlande: {Niederlande}',
 '_i10': 'a = 42\nb = 47\ndef f(): return 42',
 'a': 42,
 'b': 47,
 'f': <function __main__.f()>,
 '_i11': 'locals()'}
```

Das Dictionary, was von locals() zurückgeliegert wird, kann als Parameter für die format-Methode genutzt werden. Dadurch ist es möglich im Format-String lokale Variablen zu verwenden.

Im folgenden Beispiel benutzen wir die lokalen Variablen a, b und f:

```
print("a={a}, b={b} and f={f}".format(**locals()))
```

```
a=42, b=47 and f=<function f at 0x000001BC788F2D38>
```

Weitere String-Methoden zum Formatieren

Die String-Klasse enthält weitere Methoden, die man auch zu Formatierungszwecken nutzen kann: ljust, rjust, center und zfill.

Falls S ein String ist, können wir die vier Methoden wie folgt erklären:

- center(…):
  An den String S werden links und rechts Füllzeichen bis zur Länge “width” so angefügt, dass der ursprüngliche String zentriert wird. Wird für das Füllzeichen “fillchar” kein Wert angegeben, wird als Standardwert ein Leerzeichen verwendet.

*Beispiele:

```
s = "Python"
s.center(10)
```

```
'  Python  '
```

```
s.center(10,"*")
```

```
'**Python**'
```

- ljust(…):
  Der String S wird linksbündig zurückgeliefert. Rechts wird der String mit Füllzeichen “fillchar” bis zur Länge “width” angefüllt. Auch hier ist ein Leerzeichen der Default-Wert für “fillchar”.

*Beispiele:

```
s = "Training"
s.ljust(12)
```

```
'Training    '
```

```
s.ljust(12,":")
```

```
'Training::::'
```

- rjust(…):

Der String S wird rechtsbündig zurückgeliefert. Von links wird der String mit Füllzeichen “fillchar” bis zur Länge “width” angefüllt. Auch hier ist ein Leerzeichen der Default-Wert für “fillchar”.

*Beispiele:

```
s = "Programming"
s.rjust(15)
```

```
'    Programming'
```

```
s.rjust(15, "~")
```

```
'~~~~Programming'
```

- zfill(…):

Ein String S wird von links mit Nullen bis zu der Länge “width” aufgefüllt. Der String ”

wird nicht abgeschnitten, falls er bereits länger als die vorgesehen Länge “width” ist. Diese Methode entspricht einem S.rjust(width, “0”)

*Beispiele:

```
account_number = "43447879"
account_number.zfill(12)
```

```
'000043447879'
```

```
# can be emulated with rjust:

account_number.rjust(12,"0")
```

```
'000043447879'
```

23 Arbeiten Mit Python Dictionaries

23.0.1 Arbeiten mit Dictionaries

23.1 Einführung

Für die Bearbeitung dieses Kapitel unseres Python-Tutorials nehmen war an, dass man mit den Grundbegriffen der Python-Dictionaries und der `while`-Schleife bereits vertraut ist, so wie wir sie in den Kapitelen Dictionaries and Schleifen behandelt hatten.

Wir haben dieses Kapitel aufgenommen, um den Lernenden zusätzliche Übungen mit sowohl Dictionaries als auch `While`-Schleifen bereitzustellen. Der Fokus liegt auf verschachtelten Wörterbüchern, in unseren Beispielen auf Dictionaries mit Unterdictionaries. Aus den Erfahrungen meiner Kurse weiß ich, dass dies oft besondere Schwierigkeiten für Anfängerinnen und Anfänger verursacht.

In diesem Kapitel geht es auch um Kaffee, Tee und andere Heißgetränke, deren Konsum wir mittels Python-Dictionaries verwalten.

Im ersten Beispiel starten wir mit einem Dictionary mit drei Personen als Keys. Die Werte entsprechen der Anzahl von getrunkenen Kaffeetassen:

23.1.1 Coffee, Dictionary and a Loop

```
kaffeeliste = {"Peter": 0,
               "Eva": 0,
               "Franka": 0}

while True:
    name = input("Name: ")
    if name == "":
        break
    if name not in kaffeeliste:
        print("Diese Person kenne ich nicht!")
        continue
    kaffeeliste[name] += 1
    print(kaffeeliste[name])

print("kaffeeliste: ", kaffeeliste)
```

```
Name:  Peter

1

Name:  Petra

Diese Person kenne ich nicht!
```

```
Name:  Franka

1

Name:  Peter

2

Name:

kaffeeliste:  {'Peter': 2, 'Eva': 0, 'Franka': 1}
```

Wir dürfen natürlich nicht die Teetrinker vergessen. Deshalb erweitern wir unser Python-Programm um eine Liste für den Teekonsum:

```
kaffeeliste = {"Peter": 0,
               "Eva": 0,
               "Franka": 0}
teeliste = {"Peter": 0,
            "Eva": 0,
            "Franka": 0}

while True:
    name = input("Name: ")
    if name == "":
        break
    if name not in kaffeeliste:
        print("Diese Person kenne ich nicht!")
        continue
    getränk = input("Getränk (Kaffee/Tee): ")
    if getränk.lower() == "kaffee":
        kaffeeliste[name] += 1
        print(kaffeeliste[name])
    elif getränk.lower() == "tee":
        teeliste[name] += 1
        print(teeliste[name])
    else:
        print("Unbekanntes Getränk!")

print("Kaffeeliste: ", kaffeeliste)
print("Teeliste: ", teeliste)
```

```
Name:  Peter
Getränk (Kaffee/Tee):  Kaffee

1

Name:  Eva
Getränk (Kaffee/Tee):  Kakao

Unbekanntes Getränk!

Name:  Eva
Getränk (Kaffee/Tee):  Tee

1

Name:  Eva
Getränk (Kaffee/Tee):  Tee
```

```
2

Name:  Franka
Getränk (Kaffee/Tee):  Kaffee

1

Name:

Kaffeeliste:  {'Peter': 1, 'Eva': 0, 'Franka': 1}
Teeliste:  {'Peter': 0, 'Eva': 2, 'Franka': 0}
```

Obiger Programmierstil eignet sich nicht, wenn wir weitere Getränke aufnehmen wollen. Wir bräuchten jedesmal eine eigene Liste. Es ist bedeutend besser eine Unterliste mit den Getränken zu verwenden:

```
getränkekonsum = {"Peter": {"Tee": 0,
                            "Kaffee": 0},
                  "Eva": {"Tee": 0,
                          "Kaffee": 0},
                  "Franka": {"Tee": 0,
                             "Kaffee": 0}}

while True:
    name = input("Name: ").capitalize()
    if name == "":
        break
    if name not in getränkekonsum:    # gibt keinen Key "name"
        antwort = input("Sollen wir " + name + " in Liste aufnehmen? (j/n)")
        if antwort in ["j", "ja", "Ja", "y"]:
            getränkekonsum[name] = {"Tee": 0, "Kaffee": 0}
        else:
            print("Dann gibt's nichts zu trinken für " + name + "!")
            continue
    getränk = input("Getränk (Kaffee/Tee): ").capitalize()
    if getränk in getränkekonsum[name]:
        getränkekonsum[name][getränk] += 1
    else:
        print("Sorry, dieses Getränk führen wir nicht!")
print(getränkekonsum)
```

```
Name:  Peter
Getränk (Kaffee/Tee):  Kaffee
Name:  Eva
Getränk (Kaffee/Tee):  Tee
Name:  Peter
Getränk (Kaffee/Tee):  Kakao

Sorry, dieses Getränk führen wir nicht!

Name:  Peter
Getränk (Kaffee/Tee):  Tee
Name:  Swen
Sollen wir Swen in Liste aufnehmen? (j/n) j
Getränk (Kaffee/Tee):  Tee
Name:

{'Peter': {'Tee': 1, 'Kaffee': 1}, 'Eva': {'Tee': 1, 'Kaffee': 0}, 'Franka':
↪ {'Tee': 0, 'Kaffee': 0}, 'Swen': {'Tee': 1, 'Kaffee': 0}}
```

Das vorige Beispiel verallgemeinern wir noch weiter, um einfacher ein solches Dictionary aufbauen zu können. Das `print` haben wir eingebaut, dass man leichter verfolgen kann, wie dieses Dictionary mittels `for`-Schleifen aufgebaut wird:

```
getränke = ["Tee", "Kaffee", "Kakao", "Gemüsebrühe"]
namen = ["Peter", "Eva", "Sarah", "Eddie", "Swen"]

getränkekonsum = {}
for name in namen:
    getränkekonsum[name] = {}
    print(getränkekonsum)
    for getränk in getränke:
        getränkekonsum[name][getränk] = 0
```

```
{'Peter': {}}
{'Peter': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Eva': {}}
{'Peter': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Eva': {'Tee':
↪  0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Sarah': {}}
{'Peter': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Eva': {'Tee':
↪  0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Sarah': {'Tee': 0, 'Kaffee':
↪  0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Eddie': {}}
{'Peter': {'Tee': 0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Eva': {'Tee':
↪  0, 'Kaffee': 0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Sarah': {'Tee': 0, 'Kaffee':
↪  0, 'Kakao': 0, 'Gemüsebrühe': 0}, 'Eddie': {'Tee': 0, 'Kaffee': 0, 'Kakao':
↪  0, 'Gemüsebrühe': 0}, 'Swen': {}}
```

Aufgaben

Aufgabe 1 Gegen sei ein kleiner Supermarkt mit folgenden Produkten. Der Inhaber des Supermarktes möchte gerne wissen, wie hoch der Verkaufswert aller seiner Produkte zusammengerechnet beträgt. Schreibe dazu ein Python-Programm:

```
supermarket = {"milk": {"quantity": 20, "price": 1.19},
               "biscuits":  {"quantity": 32, "price": 1.45},
               "butter":  {"quantity": 20, "price": 2.29},
               "cheese":  {"quantity": 15, "price": 1.90},
               "bread":  {"quantity": 15, "price": 2.59},
               "cookies":  {"quantity": 20, "price": 4.99},
               "yogurt": {"quantity": 18, "price": 3.65},
               "apples":  {"quantity": 35, "price": 3.15},
               "oranges":  {"quantity": 40, "price": 0.99},
               "bananas": {"quantity": 23, "price": 1.29}}
```

Lösungen

Aufgabe 1

```
total_value = 0
for article, numbers in supermarket.items():
    quantity = numbers["quantity"]
    price = numbers["price"]
    product_price = quantity * price
    article = article + ':'
```

```
    print(f"{article:15s} {product_price:08.2f}")
    total_value += product_price
print("="*24)
print(f"Gesamtsumme:    {total_value:08.2f}")
```

```
milk:           00023.80
biscuits:       00046.40
butter:         00045.80
cheese:         00028.50
bread:          00038.85
cookies:        00099.80
yogurt:         00065.70
apples:         00110.25
oranges:        00039.60
bananas:        00029.67
========================
Gesamtsumme:    00528.37
```

24 Python3 Funktionen

24.0.1 Funktionen

Syntax

Das Konzept einer Funktion ist eines der wichtigsten in der Mathematik. Eine übliche Verwendung von Funktionen in Computersprachen ist die Implementierung mathematischer Funktionen. Eine solche Funktion berechnet ein oder mehrere Ergebnisse, die vollständig durch die an sie übergebenen Parameter bestimmt werden.

Das ist Mathematik, aber wir sprechen über Programmierung und Python. Was ist also eine Funktion in der Programmierung? Im allgemeinsten Sinne ist eine Funktion ein Strukturierungselement in Programmiersprachen, um eine Reihe von Anweisungen zu gruppieren, damit sie in einem Programm mehr als einmal verwendet werden können. Die einzige Möglichkeit, dies ohne Funktionen zu erreichen, besteht darin, Code durch Kopieren und Anpassen an verschiedene Kontexte wiederzuverwenden, was eine schlechte Idee wäre. Redundanter Code - in diesem Fall sich wiederholender Code - sollte vermieden werden! Die Verwendung von Funktionen verbessert normalerweise die Verständlichkeit und Qualität eines Programms. Dies senkt auch die Kosten für die Entwicklung und Wartung der Software.

Je nach Programmiersprache und Kontext begegnet man verwandten Begriffen wie Unterprogramm, Routine, Prozedur oder Methode. In Python bezeichnet *Methode* speziell eine Funktion, die über eine Klasse bzw. ein Objekt aufgerufen wird.

> **Moderne Funktionssignaturen:** Python kann gezielt festlegen, wie Argumente übergeben werden. Parameter vor `/` sind *positional-only*, Parameter nach einem alleinstehenden `*` *keyword-only*. Beispiel: `def skaliere(x, /, *, faktor=1): return x * faktor`. Der Aufruf `skaliere(10, faktor=2)` ist gültig; `skaliere(x=10, faktor=2)` nicht. Typannotationen wie `def addiere(a: int, b: int) -> int:` dokumentieren erwartete Typen und unterstützen Werkzeuge zur statischen Analyse, werden von Python zur Laufzeit aber nicht automatisch erzwungen.

Ein motivierendes Beispiel für Funktionen

Schauen wir uns das folgende Beispiel an:

```
print("Das Programm startet")

print("Hallo Peter")
print("Schön dich zu sehen!")
print("Viel Spaß mit dem Programm!")

# hier stelle man sich irgendwelchen Code vor:
```

```
egal = "über dies nicht nachdenken"
ist_nicht_wichtig = "normalerweise schon"

print("Hallo Dora")
print("Schön dich zu sehen!")
print("Viel Spaß mit dem Programm!")

# irgendein Code:
wie_auch_immer = "mir auch "
x = "steht stellvertretend"
y = "stellvertretend für wichtigen Code"

print("Hallo Kevin")
print("Schön dich zu sehen!")
print("Viel Spaß mit dem Programm!")
```

```
Das Programm startet
Hallo Peter
Schön dich zu sehen!
Viel Spaß mit dem Programm!
Hallo Dora
Schön dich zu sehen!
Viel Spaß mit dem Programm!
Hallo Kevin
Schön dich zu sehen!
Viel Spaß mit dem Programm!
```

Schauen wir uns den obigen Code genauer an:

Wir können im Code sehen, dass wir drei Personen begrüßen. Jedes Mal verwenden wir drei print-Aufrufe, die nahezu gleich sind. Nur der Name ist anders. Dies nennen wir redundanten Code. Wir wiederholen den Code dreimal. Das sollte natürlich nicht sein. Dies ist der Punkt, an dem Funktionen in Python verwendet werden können und sollten. Wir könnten im Code statt der Namen einen Platzhalter verwenden. Im folgenden ein Puzzleteil. Wir schreiben den Code nur noch einmal und ersetzen jeweils das Puzzleteil mit Namen:

Das war natürlich noch kein korrekter Python-Code. Wie dies in Python geht zeigen wir im folgenden Abschnitt.

Funktionen in Python

Der folgende Code verwendet eine Funktion. Das vorherige Puzzleteil ist jetzt ein Parameter mit dem Namen `name`:

```
def begruesse(name):
    print("Hallo " + name)
    print("Schön dich zu sehen!")
    print("Viel Spaß mit dem Programm!")


print("Das Programm startet")

begruesse("Peter")

# hier stelle man sich irgendwelchen Code vor:
egal = "über dies nicht nachdenken"
```

```
ist_nicht_wichtig = "normalerweise schon"

begruesse("Dora")

# irgendein Code:
wie_auch_immer = "mir auch "
x = "steht stellvertretend"
y = "stellvertretend für wichtigen Code"

begruesse("Kevin")
```

```
Das Programm startet
Hallo Peter
Schön dich zu sehen!
Viel Spaß mit dem Programm!
Hallo Dora
Schön dich zu sehen!
Viel Spaß mit dem Programm!
Hallo Kevin
Schön dich zu sehen!
Viel Spaß mit dem Programm!
```

Im vorigen Code benutzten wir eine Python-Funktion. Wir können sehen, dass eine Funktionsdefinition mit dem `def`-Schlüsselwort eingeleitet wird. Die allgemeine Syntax sieht folgendermaßen aus:

```
def funktionsname(parameterliste):
    Anweisungen, d. h. der Funktionskörper
```

Die Parameterliste besteht aus keinem oder mehreren Parametern. Parameter werden als Argumente bezeichnet, wenn die Funktion aufgerufen wird. Der Funktionskörper besteht aus eingerückten Anweisungen. Der Funktionskörper wird bei jedem Aufruf der Funktion ausgeführt. Wir zeigen dies im folgenden Bild:

Der Code aus dem Bild ist im Folgenden zu sehen:

```
def f(x, y):
    z = 2 * (x + y)
    return z


print("Programm fängt an!")
a = 3
res1 = f(a, 2+a)
print("Ergebnis des Funktionsaufrufs: ", res1)
a = 4
b = 7
res2 = f(a, b)
print("Ergebnis des Funktionsaufrufs: ", res2)
```

```
Programm fängt an!
Ergebnis des Funktionsaufrufs:  16
Ergebnis des Funktionsaufrufs:  22
```

Wir rufen die Funktion zweimal im Programm auf. Die Funktion hat zwei Parameter, die `x` und `y` heißen. Dies bedeutet, dass die Funktion `f` zwei Werte erwartet, oder ich

sollte" zwei Objekte "sagen. Zunächst rufen wir diese Funktion mit `f (a, 2 + a)` auf. Dies bedeutet, dass `a` zu `x` geht und das Ergebnis von `2 + a` (5) 'zu' der Variablen `y` geht. Der Mechanismus zum Zuweisen von Argumenten zu Parametern wird als ** Argumentübergabe ** bezeichnet. Wenn wir die Anweisung `return` erreichen, wird das von `z` referenzierte Objekt zurückgegeben, was bedeutet, dass es der Variablen `res1` zugewiesen wird. Nach Verlassen der Funktion `f` werden die Variable `z` und die Parameter `x` und `y` automatisch gelöscht.

Die Verweise auf die Objekte sind im nächsten Diagramm zu sehen:

Der nächste Python-Codeblock enthält ein Beispiel für eine Funktion ohne return-Anweisung. Wir verwenden die `pass` Anweisung innerhalb dieser Funktion. `pass` ist eine Nulloperation. Dies bedeutet, dass bei der Ausführung nichts passiert. Es ist nützlich als Platzhalter in Situationen, in denen eine Anweisung syntaktisch erforderlich ist, aber kein Code ausgeführt werden muss:

def doNothing(): pass </ pre>

Eine nützlichere Funktion:

```
def fahrenheit(T_in_celsius):
    """ Gibt die Temperatur T_in_celsius in Grad Fahrenheit zurück """
    return (T_in_celsius * 9 / 5) + 32

for t in (22.6, 25.8, 27.3, 29.8):
    print(t, ": ", fahrenheit(t))
```

```
22.6 :  72.68
25.8 :  78.44
27.3 :  81.14
29.8 :  85.64
```

" " " Gibt die Temperatur in Grad Fahrenheit zurück " " " ist der sogenannte Docstring. Er wird bei der help-Funktion verwendet:

```
help(fahrenheit)
```

```
Help on function fahrenheit in module __main__:

fahrenheit(T_in_celsius)
    Gibt die Temperatur T_in_celsius in Grad Fahrenheit zurück
```

Der Docstring hängt an dem Attribut `__doc__`:

```
fahrenheit.__doc__
```

```
' Gibt die Temperatur T_in_celsius in Grad Fahrenheit zurück '
```

Standardargumente in Python

Wenn wir eine Python-Funktion definieren, können wir einen Standardwert für einen Parameter festlegen. Wenn die Funktion ohne das Argument aufgerufen wird, wird dieser Standardwert dem Parameter zugewiesen. Dies macht einen Parameter optional. Mit anderen

Worten: Standardparameter sind Parameter, die beim Aufruf der Funktion nicht angegeben werden müssen. In diesem Fall werden die Standardwerte verwendet.

Wir werden das Funktionsprinzip von Standardparametern anhand eines einfachen Beispiels demonstrieren. Die folgende Funktion `Hallo` - die nicht sehr nützlich ist - begrüßt eine Person. Wenn kein Name angegeben wird, werden alle begrüßt:

```
def hallo(name="zusammen"):
    """ Grüßt eine Person """
    print("Hallo " + name + "!")

hallo("Peter")
hallo()
```

```
Hallo Peter!
Hallo zusammen!
```

```
def hello(name="Bruce"):
    """ Grüßt eine Person """
    print("Hello " + name + "!")

hello("Peter")
hello()
```

```
Hello Peter!
Hello Bruce!
```

Veränderliche Standardargumente

Standardwerte werden ausgewertet, **wenn die `def`-Anweisung ausgeführt und damit die Funktion definiert wird**, nicht bei jedem späteren Funktionsaufruf. Das ist besonders wichtig bei veränderbaren Objekten wie Listen oder Dictionaries.

Wird beispielsweise eine Liste direkt als Standardwert verwendet, teilen sich alle Aufrufe dieselbe Liste. Das führt häufig zu unerwarteten Ergebnissen. Wenn bei jedem Aufruf eine neue Liste entstehen soll, verwendet man üblicherweise `None` als Sentinel und erzeugt die Liste im Funktionskörper:

```
def spammer(tuete=None):
    if tuete is None:
        tuete = []
    tuete.append("spam")
    return tuete
```

Schauen wir zunächst, wie der problematische Fall mit einem veränderbaren Standardwert aussieht:

```
def spammer(tüte=[]):
    tüte.append("spam")
    return tüte
```

Wenn Sie diese Funktion einmal ohne Argument aufrufen, wird das erwartete Ergebnis zurückgegeben:

```
spammer()
```

```
['spam']
```

Die Überraschung zeigt, wenn wir die Funktion ohne Argument erneut aufrufen:

```
spammer()
```

```
['spam', 'spam']
```

Die meisten Programmierer haben das gleiche Ergebnis wie beim ersten Aufruf erwartet, d. H. "[" Spam "]" "

Um zu verstehen, was los ist, müssen Sie wissen, was passiert, wenn die Funktion definiert wird. Der Compiler erstellt ein Attribut `__defaults__`:

```
def spammer(tüte=[]):
    tüte.append("spam")
    return tüte

spammer.__defaults__
```

```
([],)
```

Wann immer wir die Funktion aufrufen, wird der Parameter `bag` dem Listenobjekt zugewiesen, auf das `Spammer .__ Defaults __ [0]` verweist:

```
for i in range(5):
    print(spammer())

print("spammer.__defaults__", spammer.__defaults__)
```

```
['spam']
['spam', 'spam']
['spam', 'spam', 'spam']
['spam', 'spam', 'spam', 'spam']
['spam', 'spam', 'spam', 'spam', 'spam']
spammer.__defaults__ (['spam', 'spam', 'spam', 'spam', 'spam'],)
```

Jetzt wissen und verstehen Sie, was los ist, aber Sie fragen sich vielleicht, wie Sie dieses Problem lösen können. Die Lösung besteht darin, den unveränderlichen Wert `None` als Standard zu verwenden. Auf diese Weise kann die Funktion bag dynamisch (zur Laufzeit) auf eine leere Liste setzen:

```
def spammer(tüte=None):
    if tüte is None:
        tüte = []
    tüte.append("spam")
    return tüte

for i in range(5):
    print(spammer())
```

```
print("spammer.__defaults__", spammer.__defaults__)
```

```
['spam']
['spam']
['spam']
['spam']
['spam']
spammer.__defaults__ (None,)
```

Docstrings

Ein *Docstring* ist eine String-Literal-Anweisung als erste Anweisung im Funktionskörper. Python speichert diesen Dokumentationstext im Attribut `funktion.__doc__`; Werkzeuge wie `help()` können ihn anzeigen.

Beispiel:

```
def hallo(name="zusammen"):
    """ Grüßt einer Person """
    print("Hallo " + name + "!")

print("Die Docstring der Funktion Hallo:" + hallo.__doc__)
```

```
Die Docstring der Funktion Hallo: Grüßt einer Person
```

Schlüsselwortargumente

Bei einem Funktionsaufruf können viele Parameter entweder über ihre Position oder explizit über ihren Namen belegt werden. Ein benannt übergebener Wert heißt *keyword argument* (Schlüsselwortargument). Dadurch können Aufrufe lesbarer werden und optionale Parameter gezielt gesetzt werden.

Ein Beispiel:

```
def sumsub(a, b, c=0, d=0):
    return a - b + c - d

print(sumsub(12, 4))
print(sumsub(42, 15, d=10))
```

```
8
17
```

Ein Parameter darf bei einem Aufruf nicht gleichzeitig positional **und** per Schlüsselwort belegt werden; ansonsten erhält er zwei Werte und Python meldet einen `TypeError`. Abgesehen von ausdrücklich *positional-only* deklarierten Parametern können normale Parameter aber grundsätzlich auch per Schlüsselwort übergeben werden.

Im Beispiel können wir `d` gezielt setzen und für `c` den Standardwert beibehalten. Ohne Schlüsselwortargument müssten wir den Wert für `c` ausdrücklich angeben:

```
print(sumsub(42,15,0,10))
```

```
17
```

Rückgabewerte

In unseren vorherigen Beispielen haben wir eine return-Anweisung in der Funktion sumsub verwendet, jedoch nicht in Hello. Wir können also sehen, dass eine return-Anweisung nicht zwingend erforderlich ist. Aber was wird zurückgegeben, wenn wir nicht explizit eine return-Anweisung geben. Mal schauen:

```
def no_return(x, y):
    c = x + y

res = no_return(4, 5)
print(res)
```

```
None
```

Wenn wir dieses kleine Skript starten, wird *None* gedruckt, d. H. Der spezielle Wert *None* wird von einer Funktion ohne Rückgabe zurückgegeben. *None* wird auch zurückgegeben, wenn wir nur eine Rückgabe in einer Funktion ohne Ausdruck haben:

```
def empty_return(x, y):
    c = x + y
    return

res = empty_return(4, 5)
print(res)
```

```
None
```

Andernfalls wird der Wert des Ausdrucks nach der Rückgabe zurückgegeben. Im nächsten Beispiel wird 9 gedruckt:

```
def return_sum(x, y):
    c = x + y
    return c

res = return_sum(4, 5)
print(res)
```

```
9
```

Fassen wir dieses Verhalten zusammen: Funktionskörper können eine oder mehrere return-Anweisungen enthalten. Sie können sich überall im Funktionskörper befinden. Eine return-Anweisung beendet die Ausführung des Funktionsaufrufs und "gibt" das Ergebnis, d. H. Den Wert des Ausdrucks nach dem Schlüsselwort return, an den Aufrufer zurück. Wenn die return-Anweisung keinen Ausdruck enthält, wird der spezielle Wert `None` zurückgegeben. Wenn der Funktionscode keine return-Anweisung enthält, endet die Funktion, wenn der Kontrollfluss das Ende des Funktionskörpers erreicht und der Wert `None` zurückgegeben wird.

Mehrere Werte zurückgeben

Eine Funktion kann genau einen Wert zurückgeben, oder wir sollten besser ein Objekt sagen. Ein Objekt kann ein numerischer Wert sein, z. B. eine Ganzzahl oder ein Gleitkommawert. Es kann aber auch z.B. eine Liste oder ein Wörterbuch. Wenn wir also beispielsweise 3 Ganzzahlwerte zurückgeben müssen, können wir eine Liste oder ein Tupel mit diesen drei Ganzzahlwerten zurückgeben. Das heißt, wir können indirekt mehrere Werte zurückgeben. Das folgende Beispiel, das die Fibonacci-Grenze für eine positive Zahl berechnet, gibt ein 2-Tupel zurück. Das erste Element ist die größte Fibonacci-Zahl kleiner als x und die zweite Komponente ist die kleinste Fibonacci-Zahl größer als x. Der Rückgabewert wird sofort durch Auspacken in die Variablen lub und sup gespeichert:

```
def fib_intervall(x):
    """ gibt die größten Fibonacci zurück
     Zahl kleiner als x und die niedrigste
     Fibonacci-Zahl höher als x"""
    if x < 0:
        return -1
    (alt,neu) = (0,1)
    while True:
        if neu < x:
            (alt,neu) = (neu,alt+neu)
        else:
            if neu == x:
                neu = alt+neu
            return (alt, neu)

while True:
    x = int(input("Ihre Nummer: "))
    if x <= 0:
        break
    (lub, sup) = fib_intervall(x)
    print("Größte Fibonacci-Zahl kleiner als x: " + str(lub))
    print("Kleinste Fibonacci-Zahl größer als x: " + str(sup))
```

```
Ihre Nummer:  5990

Größte Fibonacci-Zahl kleiner als x: 4181
Kleinste Fibonacci-Zahl größer als x: 6765

Ihre Nummer:  0
```

Lokale und globale Variablen in Funktionen

Variablennamen sind standardmäßig lokal für die Funktion, in der sie definiert werden.

```
def f():
    print(s)
s = "Python"
f()
```

```
Python
```

```
def f():
```

```
    s = "Perl"
    print(s)
f()
```

```
Perl
```

```
s = "Python"
f()
print(s)
```

```
Perl
Python
```

```
def f():
    print(s)
    s = "Perl"
    print(s)


s = "Python"
f()
print(s)
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-64-81b2fbbc4d42> in <module>
      6
      7 s = "Python"
----> 8 f()
      9 print(s)

<ipython-input-64-81b2fbbc4d42> in f()
      1 def f():
----> 2     print(s)
      3     s = "Perl"
      4     print(s)
      5

UnboundLocalError: local variable 's' referenced before assignment
```

Die Fehlermeldung entsteht, weil Python bereits beim Kompilieren des Funktionskörpers erkennt, dass `s` innerhalb von `f()` zugewiesen wird. Ohne eine `global`- oder `nonlocal`-Deklaration gilt `s` deshalb in der gesamten Funktion als lokaler Name. Beim ersten `print(s)` besitzt diese lokale Variable aber noch keinen Wert, daher entsteht ein `UnboundLocalError`.

```
def f():
    global s
    print(s)
    s = "Hund"
    print(s)
s = "Katze"
f()
print(s)
```

```
Katze
```

```
Hund
Hund
```

Wir haben die Variable innerhalb des Skripts global gemacht. Daher wird alles, was wir innerhalb des Funktionskörpers von f mit s tun, mit der globalen Variablen s außerhalb von f gemacht.

Beliebige Anzahl von Parametern

Es gibt viele Situationen in der Programmierung, in denen die genaue Anzahl der erforderlichen Parameter nicht a priori bestimmt werden kann. In Python kann eine variable Anzahl positionaler Argumente mit `*args` aufgenommen werden. Innerhalb der Funktion ist `args` ein Tupel. Ein Sternchen `*` steht dabei vor dem Parameternamen. Dieses Sternchen sollte nicht mit der C-Syntax verwechselt werden, bei der diese Notation mit Zeigern verbunden ist. Beispiel:

```
def arithmetisches_mittel(erste, *werte):
    """ Diese Funktion berechnet das arithmetische Mittel eines nicht leeren
        beliebige Anzahl von Zahlenwerten """

    return (erste + sum(werte)) / (1 + len(werte))

print(arithmetisches_mittel(45,32,89,78))
print(arithmetisches_mittel(8989.8,78787.78,3453,78778.73))
print(arithmetisches_mittel(45,32))
print(arithmetisches_mittel(45))
```

```
61.0
42502.3275
38.5
45.0
```

Das ist großartig, aber wir haben immer noch ein Problem. Möglicherweise haben Sie eine Liste mit numerischen Werten. Wie zum Beispiel,

Sie können es nicht mit anrufen

weil “arithmetisches_mittel” mit einer Liste nicht fertig wird. Ich nenne es mit

ist umständlich und vor allem innerhalb eines Programms unmöglich, da die Liste beliebig lang sein kann.

Die Lösung ist einfach. Wir fügen einen Stern vor dem x hinzu, wenn wir die Funktion aufrufen.

Dadurch wird die Liste “entpackt” oder vereinzelt.

Ein praktisches Beispiel: Wir haben eine Liste von 4, 2-Tupel-Elementen:

Wir möchten diese Liste in die folgende Liste mit 2 Elementen und 4 Tupeln umwandeln:

Dies kann mithilfe des * -Operators und der Zip-Funktion folgendermaßen erfolgen:

Liste (zip (* meine_liste)) </ pre>

Beliebige Anzahl von Schlüsselwortparametern

Im vorherigen Kapitel haben wir gezeigt, wie eine beliebige Anzahl von Positionsparametern an eine Funktion übergeben wird. Es ist auch möglich, eine beliebige Anzahl von Schlüsselwortparametern an eine Funktion als Wörterbuch zu übergeben. Zu diesem Zweck müssen wir das doppelte Sternchen "**" verwenden.

```
def f(**kwargs):
    print(kwargs)
```

```
f()
```

```
{}
```

```
f(de="Deutsch",en="Englisch",fr="Französisch")
```

```
{'de': 'Deutsch', 'en': 'Englisch', 'fr': 'Französisch'}
```

Ein Anwendungsfall ist der folgende:

```
def f(a, b, x, y):
    print(a, b, x, y)
d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
f(**d)
```

```
append block extract yes
```

Übungen mit Funktionen

Übung 1 Im Kapitel „Bedingte Anweisungen hatten wir ein Beispielprogramm, in dem wir ein Hundealter in Menschenalter mit folgenden Regeln umgerechnet hatten:

- Ein einjähriger Hund entspricht in etwa einem 14-jährigen Menschen
- Ein Hund der zwei Jahre alt ist, entspricht entwicklungsmäßig einem 22-jährigen Menschen.
- Jedes weitere Hundejahr entspricht fünf Menschenjahre.

Schreiben Sie nun eine Funktion für diese Umwandlung.

Übung 2 Schreiben Sie eine Funktion, die einen Text aufnimmt und mit einer Caesar-Chiffre verschlüsselt. Dies ist eine der einfachsten und am häufigsten bekannten Verschlüsselungstechniken. Jeder Buchstabe im Text wird durch einen Buchstaben ersetzt, der eine feste Anzahl von Stellen weiter im Alphabet enthält.

Was ist mit dem Entschlüsseln des codierten Textes?

Die Caesar-Chiffre ist eine Substitutions-Chiffre.

Übung 3 Wir können eine weitere Substitutions-Chiffre erstellen, indem wir den Propheten permutieren und die Buchstaben dem entsprechenden permutierten Alphabet zuordnen.

Schreiben Sie eine Funktion, die einen Text und ein Wörterbuch benötigt, um den angegebenen Text mit einem permutierten Alphabet zu entschlüsseln oder zu verschlüsseln.

Übung 4 Schreiben Sie eine Funktion txt2morse, die einen Text in Morsecode übersetzt, d. H. Die Funktion gibt eine Zeichenfolge mit dem Morsecode zurück.

Schreiben Sie eine weitere Funktion morse2txt, die eine Zeichenfolge im Morsecode in eine „normale“ Zeichenfolge übersetzt.

Die Morsezeichen sind durch Leerzeichen getrennt. Wörter mit drei Leerzeichen.

Übung 5 Vielleicht ist der erste Algorithmus, der zur Approximation von $ sqrt {S} $ verwendet wird, als “babylonische Methode” bekannt, benannt nach den Babyloniern, oder “Heldenmethode”, benannt nach dem griechischen Mathematiker Hero of Alexandria aus dem ersten Jahrhundert, der den ersten expliziten gab Beschreibung der Methode.

Wenn eine Zahl $ x_n $ nahe an der Quadratwurzel von $ a $ liegt, dann

$$x_{n+1} = frac12(x_n + fracax_n)$$

wird eine bessere Annäherung sein.

Schreiben Sie ein Programm, um die Quadratwurzel einer Zahl nach der babylonischen Methode zu berechnen.

Übung 6 Schreiben Sie eine Funktion, die die Position des n-ten Auftretens von a berechnet String Sub in einem anderen String s. Wenn sub in s nicht vorkommt, wird -1 zurückgegeben.

Lösungen

Lösung zu Übung 1

```
def dog_age2human_age(dog_age):
    """dog_age2human_age(dog_age)"""
    if dog_age == 1:
        human_age = 14
    elif dog_age == 2:
        human_age = 22
    else:
        human_age =  22 + (dog_age-2)*5
    return human_age

age = int(input("Wie alt ist dein Hund? "))
print(f"Enspricht {dog_age2human_age(age)} Menschenjahren")
```

```
Wie alt ist dein Hund?  5

Enspricht 37 Menschenjahren
```

Lösung zu Übung 2

```
import string
```

```
abc = string.ascii_uppercase
def caesar(txt, n, coded=False):
    """ Gibt den codierten oder decodierten Text zurück """
    result = ""
    for char in txt.upper():
        if char not in abc:
            result += char
        elif coded:
            result += abc[(abc.find(char) + n) % len(abc)]
        else:
            result += abc[(abc.find(char) - n) % len(abc)]
    return result

n = 3
x = caesar("Hallo, hier bin ich!", n)
print(x)
print(caesar(x, n, True))
```

```
EXIIL, EFBO YFK FZE!
HALLO, HIER BIN ICH!
```

In der vorherigen Lösung ersetzen wir nur die Buchstaben. Jedes Sonderzeichen bleibt unberührt. Die folgende Lösung fügt einige Sonderzeichen hinzu, die ebenfalls permutiert werden. Spezielle Zeichen, die nicht in abc enthalten sind, gehen bei dieser Lösung verloren!

```
import string
abc = string.ascii_uppercase + " .,-?!"
def caesar(txt, n, coded=False):
    """ Gibt den codierten oder decodierten Text zurück """
    result = ""
    for char in txt.upper():
        if coded:
            result += abc[(abc.find(char) + n) % len(abc)]
        else:
            result += abc[(abc.find(char) - n) % len(abc)]
    return result

n = 3
x = caesar("Hallo, hier bin ich!", n)
print(x)
print(caesar(x, n, True))
```

```
E-IILZXEFBOX?FKXF!E,
HALLO, HIER BIN ICH!
```

Lösung zu Übung 3

```
import string
from random import sample

alphabet = string.ascii_letters
permutated_alphabet = sample(alphabet, len(alphabet))

encrypt_dict = dict(zip(alphabet, permutated_alphabet))
decrypt_dict = dict(zip(permutated_alphabet, alphabet))
```

```
def encrypt(text, edict):
    """ Every character of the text 'text'
    is mapped to the value of edict. Characters
    which are not keys of edict will not change"""
    res = ""
    for char in text:
        res = res + edict.get(char, char)
    return res

# Donald Trump: 5:19 PM, September 9 2014
txt = """Windmills are the greatest
threat in the US to both bald
and golden eagles. Media claims
fictional ‘global warming’ is worse."""

ctext = encrypt(txt, encrypt_dict)
print(ctext + "\n")
print(encrypt(ctext, decrypt_dict))
```

```
XObLNOvvc eqB diB ZqBedBcd
diqBed Ob diB IP dj Ejdi EevL
ebL ZjvLBb BeZvBc. zBLOe oveONc
VOodOjbev ‘ZvjEev neqNObZ’ Oc njqcB.

Windmills are the greatest
threat in the US to both bald
and golden eagles. Media claims
fictional ‘global warming’ is worse.
```

Lösung zu Übung 4

```
latin2morse_dict = {'A':'.-', 'B':'-...', 'C':'-.-.', 'D':'-..',
                    'E':'.', 'F':'..-.', 'G':'--.','H':'....',
                    'I':'..', 'J':'.---', 'K':'-.-', 'L':'.-..',
                    'M':'--', 'N':'-.', 'O':'---', 'P':'.--.',
                    'Q':'--.-', 'R':'.-.', 'S':'...', 'T':'-',
                    'U':'..-', 'V':'...-', 'W':'.--', 'X':'-..-',
                    'Y':'-.--', 'Z':'--..', '1':'.----', '2':'...--',
                    '3':'...--', '4':'....-', '5':'.....', '6':'-....',
                    '7':'--...', '8':'---..', '9':'----.', '0':'-----',
                    ',':'--..--', '.':'.-.-.-', '?':'..--..', ';':'-.-.-',
                    ':':'---...', '/':'-..-.', '-':'-....-', '\'':'.----.',
                    '(':'-.--.-', ')':'-.--.-', '[':'-.--.-', ']':'-.--.-',
                    '{':'-.--.-', '}':'-.--.-', '_':'..--.-'}

# reversing the dictionary:
morse2latin_dict = dict(zip(latin2morse_dict.values(),
                            latin2morse_dict.keys()))

print(morse2latin_dict)
```

```
{'.-': 'A', '-...': 'B', '-.-.': 'C', '-..': 'D', '.': 'E', '..-.': 'F', '--.':
↪  'G', '....': 'H', '..': 'I', '.---': 'J', '-.-': 'K', '.-..': 'L', '--': 'M',
↪  '-.': 'N', '---': 'O', '.--.': 'P', '--.-': 'Q', '.-.': 'R', '...': 'S', '-':
↪  'T', '..-': 'U', '...-': 'V', '.--': 'W', '-..-': 'X', '-.--': 'Y', '--..':
↪  'Z', '.----': '1', '...--': '3', '....-': '4', '.....': '5', '-....': '6',
↪  '--...': '7', '---..': '8', '----.': '9', '-----': '0', '--..--': ',',
↪  '.-.-.-': '.', '..--..': '?', '-.-.-': ';', '---...': ':', '-..-.': '/',
↪  '-....-': '-', '.----.': "'", '-.--.-': '}', '..--.-': '_'}
```

```
def txt2morse(txt, alphabet):
    morse_code = ""
    for char in txt.upper():
        if char == " ":
            morse_code += "    "
        else:
            morse_code += alphabet[char] + " "
    return morse_code

def morse2txt(txt, alphabet):
    res = ""
    mwords = txt.split("    ")
    for mword in mwords:
        for mchar in mword.split():
            res += alphabet[mchar]
        res += " "
    return res

mstring = txt2morse("So was?", latin2morse_dict)
print(mstring)
print(morse2txt(mstring, morse2latin_dict))
```

```
... ---    .-- .- ... ..--..
SO WAS?
```

Lösung zu Übung 5

```
def heron(a, eps=0.000000001):
    """ Approximate the square root of a"""
    previous = 0
    new = 1
    while abs(new - previous) > eps:
        previous = new
        new = (previous + a/previous) / 2
    return new

print(heron(2))
print(heron(2, 0.001))
```

```
1.414213562373095
1.4142135623746899
```

Lösung zu Übung 6

```
def findnth(s, sub, n):
```

```
    num = 0
    start = -1
    while num < n:
        start = s.find(sub, start+1)
        if start == -1:
            break
        num += 1

    return start

s = "abc xyz abc jkjkjk abc lkjkjlkj abc jlj"
print(findnth(s,"abc", 3))
```

```
19
```

25 Python3 Parameter

25.0.1 Parameterübergabe: Parameter und Argumente

Eine Funktion oder eine Prozedur benötigt üblicherweise Information über die Umgebung aus der heraus sie aufgerufen worden ist. Die Schnittstelle zwischen dieser Umgebung und der Funktion bzw. Prozedur, d.h. dem Funktionsrumpf (body) bzw. Prozedurrumpf, besteht aus speziellen Variablen, die man als Parameter bezeichnet. Indem man Parameter benutzt, kann man verschiedenste Objekte von "außen" innerhalb der Funktion benutzen. Die Syntax, wie man Parameter deklariert, und die Semantik, wie man die Argumente des Aufrufs an die formalen Parameter der Funktion oder Prozedur weiterleitet sind abhängig von der jeweiligen Programmiersprache.

Im vorigen Abschnitt haben wir mehrfach die Begriffe Parameter und Argumente verwendet. Bei einigen Lesern hat sich sicherlich der Eindruck eingestellt, dass die beiden Begriffe synonym sind. Auch in der Literatur werden diese Begriffe häufig synonym verwendet. Dennoch gibt es einen klar definierten Unterschied. Parameter stehen im Kopf der Funktion oder Prozedur, also in der Definition der Funktion oder Prozedur. Während Argumente beim Aufruf verwendet werden. Sie liefern die Werte für die formalen Parameter während der Laufzeit des Programmes.

Wertübergabe oder Referenzübergabe

Die Auswertungsstrategie für Argumente, das heißt wie die Argumente eines Funktionsaufrufes an die formalen Parameter der Funktion übergeben werden, unterscheidet sich von Programmiersprache zu Programmiersprache. Die häufigsten Strategien sind die Wertübergabe (englisch: call by value) und Referenzübergabe (auch Variablenübergabe, englisch: call by reference)

- Wertübergabe (call by value) Bei der Wertübergabe stellt beim Aufruf einer Funktion jedes Argument einen Wert dar, der ausgewertet wird. Dieser Wert wird dann an den entsprechenden formalen Parameter übergeben, d.h. der Wert des i-ten Ausdrucks (Parameter) wird beim Aufruf dem i-ten Parameter zugewiesen. Die Ausdrücke können aus Literalen, Variablen oder beliebigen Ausdrücke (wie z.B. "a + b + 7" bestehen. Werte von übergebenen Parametern können nicht verändert werden!
- Referenzübergabe(call by reference) Bei der Referenzübergabe werden, wie der Name impliziert, Referenzen (Speicheradressen) übergeben, d.h. die Speicheradresse des jeweiligen tatsächlichen Parameters. Der formale Parameter wird also zu einem Zeiger auf die Speicheradresse des aufrufenden Parameters. Das bewirkt, dass alle Änderungen an diesem Parameter innerhalb der Funktion sich auch im aufrufenden Teil des Programmes auswirken. Das bedeutet aber auch, dass die tatsächlichen Parameter beim Aufruf nur Ausdrücke sein können, deren Adresse berechnet werden können, also Variablen.

Anmerkung: Die Namensparameter (call by name), wie sie in ALGOL 60 und COBOL genutzt wurde, ist heute nicht mehr üblich.

und wie sieht es bei Python aus?

In vielen Büchern oder Einführungen in Python liest man, dass Python den einen oder den anderen Übergabemechanismus habe, also "call by reference" oder "call by value". Was ist nun richtig?

Die Erklärung liefert uns Humpty Dumpty:

"Wenn ich ein Wort verwende", erwiderte Humpty Dumpty ziemlich geringschätzig, "dann bedeutet es genau, was ich es bedeuten lasse, und nichts anderes."

"Die Frage ist doch", sagte Alice, "ob du den Worten einfach so viele verschiedene Bedeutungen geben kannst".

"Die Frage ist", sagte Humpty Dumpty, "wer die Macht hat - und das ist alles. [...]"

Lewis Carroll, Through the Looking-Glass (Alles hinter den Spiegeln)

Doch was hat das mit unserer ursprünglichen Frage zu tun: Die Autoren "dehnen" die Begriffe "call-by-value" und "call-by-reference", so dass sie passen, also "dann bedeutet es genau, was ich es bedeuten lasse, und nichts anderes."

Python benutzt einen Mechanismus, den man als "Call-by-Object" (auch "Call by Object Reference" oder "Call by Sharing") bezeichnet.

Wenn man unveränderliche Argumente wie Integers, Strings oder Tuples an eine Funktion übergibt, verhält sich die Übergabe nach außen hin wie eine Wertübergabe. Die Referenz auf das unveränderliche Objekt wird an den formalen Parameter der Funktion übertragen. Innerhalb der Funktion kann der Inhalt des Objektes nicht verändert werden, da es sich ja um unveränderliche Objekte handelt.

Anders verhält es sich jedoch, wenn man veränderliche Argumente überträgt. Auch sie werden per Referenz an die Funktionsparameter übertragen. Allerdings können einzelne Elemente eines solchen veränderlichen Objektes im Funktionsrumpf verändert werden. Wenn wir beispielsweise eine Liste an eine Funktion übergeben, müssen wir zwei Fälle unterscheiden: Die Elemente einer Liste können verändert werden, d.h. diese Änderung wirkt sich auch auf den Gültigkeitsbereich aus, aus dem die Funktion aufgerufen wurde. Wird allerdings ein komplett neues Element dem formalen Parameter zugewiesen, z.B. eine andere Liste, dann hat dies keine Auswirkungen auf die "alte" Liste, also auf die Liste, die beim Aufruf übergeben worden ist.

Zuerst wollen wir einen Blick auf die Integer-Variablen werfen. Der Parameter innerhalb der Funktion ref_demo bleibt solange eine Referenz auf das Objekt, das übergeben wurde, wie keine Änderung am Parameter erfolgt. Sobald also ein neuer Wert an den Parameter überwiesen wird, erzeugt Python eine neue Speicherstelle für das Objekt und der formale Parameter zeigt nun auf diese Speicherstelle. Die Variable des Funktionsaufrufes wird dadurch nicht verändert, wie wir im folgenden Beispiel nachvollziehen können:

```
def ref_demo(x):
    print("x=",x," id=",id(x))
    x=42
    print("x=",x," id=",id(x))
```

In dem obigen Beispiel haben wir die Identitätsfunktion id() benutzt, die ein Objekt als Parameter hat. id(obj) liefert die "Identität" des Objektes obj. Diese Identität, der Rückgabewert der Funktion, ist ein Integer, der eindeutig und konstant für dieses Objekt

ist, solange es im Programmablauf existiert.

Zwei Objekte, die gleichzeitig existieren müssen verschiedene Identitäten haben. Zwei verschiedene Objekte, die nicht-überlappende Lebensbereiche haben, dürfen allerdings den gleichen Identitätswert haben.

Wenn man die Funktion ref_demo() des obigen Beispiels aufruft, so wie wir es im grünen Block weiter unten tun, können wir prüfen, was mit x passiert: Wir können im Hauptprogramm (main Gültigkeitsbereich) sehen, dass x die Identität zBs. 41902552 hat. In der ersten print-Anweisung der ref_demo()-Funktion, wird das x aus dem main-Gültigkeitsbereich benutzt, denn wir sehen, dass wir den gleichen Identitätswert für x erhalten. Wenn wir jedoch den Wert 42 dem x zuweisen, erhält x eine neue Identität, zBs. 41903752, das heißt eine separate Speicherstelle und das x in main bleibt unberührt, d.h. es hat weiterhin den Wert 9.

Dies bedeutet, dass sich Python zuerst wie Call-by-Reference verhält, aber sobald es einen Wert zugewiesen bekommen, verhält es sich wie bei Call-by-Value. Es wird also eine lokale Speicherposition für den formalen Parameter x angelegt und das x in main behält seinen ursprünglichen Wert:

```
x = 9
id(x)
```

```
140728967733904
```

```
ref_demo(x)
```

```
x= 9  id= 140728967733904
x= 42  id= 140728967734960
```

```
id(x)
```

```
140728967733904
```

Seiteneffekte

Von einem Funktionsaufruf erwartet man, dass die Funktion den korrekten Wert für die Argumente zurückliefert und sonst keine Effekte verursacht, z.B. Ändern von Speicherzuständen. Auch wenn manche Programmierer bewusst Seiteneffekte zur Lösung ihrer Probleme einsetzen, wird dies im Allgemeinen als schlechter Stil betrachtet. Schlimmer ist es jedoch, wenn Seiteneffekte auftreten, mit denen man nicht gerechnet hat, und dadurch Fehler verursacht werden. Dies kann auch in Python passieren. Dies kann dann geschehen, wenn Listen oder Dictionaries als Parameter übergeben werden. Im folgenden Beispiel wird die Liste fib, die als aktueller Parameter an die Funktion s() übergeben wird, mit der Liste [47,11] verkettet.

```
def s(liste):
    print(id(liste))
    liste += [47,11]
    print(id(liste))
```

```
fib = [0,1,1,2,3,5,8]
id(fib)
```

```
2364656585160
```

```
s(fib)
```

```
2364656585160
2364656585160
```

```
id(fib)
```

```
2364656585160
```

```
fib
```

```
[0, 1, 1, 2, 3, 5, 8, 47, 11]
```

Man kann dies verhindern, wenn man statt einer Liste eine Kopie der Liste als Parameter übergibt. Mittels der Slicing Funktion kann man ganz leicht eine Kopie erzeugen. So wird in s(fib[:]) nicht die Liste fib sonder eine komplette Kopie übergeben. Der Aufruf verändert also nicht den Wert von fib, wie man im folgenden interaktiven Programmstück sehen kann:

```
def s(liste):
     liste += [47,11]
     print(liste)

fib = [0,1,1,2,3,5,8]
s(fib[:])
```

```
[0, 1, 1, 2, 3, 5, 8, 47, 11]
```

```
fib
```

```
[0, 1, 1, 2, 3, 5, 8]
```

Übergabe von Argumenten

Nicht nur Funktionen, sondern auch einem Python-Skript kann man Argumente übergeben. Man bezeichnet diese als Kommandozeilenparameter. Ruft man ein Python-Skript aus einer Shell auf, werden die Argumente jeweils durch ein Leerzeichen voneinander getrennt hinter dem Skriptnamen aufgeführt. Im Python-Skript selber sind die Argumente, also die Kommandozeilenparameter, als Liste unter dem Namen sys.argv abrufbar. Zusätzlich zu den Kommandozeilenparametern enthält diese Liste auch den Namen des aufrufenden Skriptes (Dateiname). Dieser steht als erster Eintrag in der Liste, also sys.argv[0]. Das Skript (argumente.py) listet sämtliche mitgegebenen Argumente in der Standardausgabe auf:

Beispiel eines Aufrufes, falls das Skript unter argumente.py gespeichert worden ist:

Dieser Aufruf erzeugt folgende Ausgabe

argumente.py python kurs fuer anfaenger

Variable Anzahl von Parametern

Wir führen nun Funktionen ein, die mit einer beliebige Anzahl von Argumenten aufgerufen werden können. Diejenigen mit Programmiererfahrungen in C und C++ kennen das Konzept als varargs.

Es folgen einige Definitionen, die nicht wichtig für das Verständnis der weiteren Beispiele dieses Kapitels sind: Der Begriff Stelligkeit oder Arität bezeichnet in der Informatik die Parameteranzahl von Funktionen, Prozeduren oder Methoden.

Als variadische Funktion bezeichnet man in der Informatik Funktionen, Prozeduren oder Methoden mit unbestimmter Arität, also solche, deren Parameteranzahl nicht bereits in ihrer Deklaration festgelegt ist.

Ein Sternchen "*" wird in Python benutzt, um eine variable Anzahl von Parametern zu kennzeichnen. Dazu wird das Sternchen unmittelbar vor einem Variablennamen gestellt.

```
def varpafu(*x): print(x)
varpafu()
varpafu(34,"Magst du Python?", "Natürlich")
```

```
()
(34, 'Magst du Python?', 'Natürlich')
```

Aus dem vorigen Beispiel lernen wir, dass die Argumente, die bei einem Funktionsaufruf von varpafu übergeben werden, in einem Tupel gesammelt werden. Auf dieses Tupel kann als "normale" Variable im Rumpf (body) der Funktion zugegriffen werden. Ruft man die Funktion ohne jegliche Argumente auf, so ist x ein leeres Tupel.

Manchmal ist es notwendig, dass man eine feste Anzahl von Positionsparametern benötigt, die von einer beliebigen Anzahl von Parametern gefolgt werden können. Dies ist möglich, aber die Positionsparameter müssen immer zuerst kommen.

Im folgenden Beispiel benutzen wir einen Positionsparameter "city", der immer angegeben werden muss, gefolgt von einer beliebigen Anzahl von weiteren Städten "andere_städte":

```
def standorte(stadt, *andere_städte): print(stadt, andere_städte)
standorte("Berlin")
standorte("Berlin","Freiburg","Stuttgart","Konstanz","Frankfurt")
```

```
Berlin ()
Berlin ('Freiburg', 'Stuttgart', 'Konstanz', 'Frankfurt')
```

"*" in Funktionsaufrufen

Ein Stern kann auch in einem Funktionsaufruf erscheinen, wie wir in der vorigen Übung gesehen haben: Die Semantik ist in diesem Fall "invers" zu der Verwendung eines Sternes in der Funktionsdefinition: Ein Argument wird entpackt und nicht gepackt. In anderen Worten: Die Elemente einer Liste oder eines Tuples werden vereinzelt:

```
def f(x,y,z):
     print(x,y,z)

p = (47,11,12)
f(*p)
```

```
47 11 12
```

Es besteht wohl kaum die Notwendigkeit zu erwähnen, dass diese Art unsere Funktion aufzurufen komfortabler als die folgende ist:

```
f(p[0],p[1],p[2])
```

```
47 11 12
```

Zusätzlich, dass dieser Aufruf weniger komfortabel ist, ist die vorige Aufrufsart im allgemeinen Fall nicht anwendbar, d.h. wenn Listen "unbekannter" Längen verwendet werden sollen. "Unbekannt" bedeutet, dass die Länge erst während der Laufzeit bekannt ist und nicht während man das Skript schreibt. Beliebige Schlüsselwortparameter Es gibt auch einen Mechanismus für eine beliebige Anzahl von Schlüsselwortparametern. Um dies zu ermöglichen wurde als Notation ein doppeltes Sternchen "**" eingeführt:

```
def f(**args):
     print(args)

f()
f(de="deutsch",en="englisch",fr="französisch")
```

```
{}
{'de': 'deutsch', 'en': 'englisch', 'fr': 'französisch'}
```

Doppeltes Sternchen im Funktionsaufruf

Das folgende Beispiel veranschaulicht die Verwendung von ** in einem Funktionsaufruf:

```
def f(a,b,x,y):
     print(a,b,x,y)

d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
f(**d)
```

```
append block extract yes
```

und jetzt in Kombination mit *:

```
t = (47,11)
d = {'x':'extract','y':'yes'}
f(*t, **d)
```

```
47 11 extract yes
```

Übung Schreibe eine Funktion, die das arithmetische Mittel aus einer variablen Anzahl von Werten berechnet. #### Lösung

```
def arithmetisches_Mittel(x, *l):
    """ Die Funktion berechnet das arithmetische Mittel eines nicht leeren
        beliebige Anzahl von Zahlen """
    sum = x
    for i in l:
        sum += i

    return sum / (1.0 + len(l))
```

Man mag sich fragen, warum wir sowohl einen Positionsparameter "x" und einen Parameter für eine variable Anzahl von Werten "*l" in unserer Funktionsdefinition benutzt haben. Die Idee besteht darin, dass wir erzwingen wollten, dass unsere Funktion immer mit einer nichtleeren Anzahl von Argumenten aufgerufen wird. Dies ist notwendig, um eine Division durch 0 zu vermeiden, was einen Fehler verursachen würde.

In der folgenden interaktiven Python-Sitzung, lernen wir, wie wir diese Funktion benutzen können. Dazu nehmen wir an, dass die Funktion arithmetic_mean in einer Datei mit dem Namen statistics.py gespeichert worden ist.

```
arithmetisches_Mittel(4,7,9)
arithmetisches_Mittel(4,7,9,45,-3.7,99)
```

```
26.71666666666667
```

Das funktioniert gut, aber die Sache hat einen Haken. Was, wenn jemand die Funktion mit einer Liste statt mit einer variablen Zahl von Zahlen aufrufen will?

Im Folgenden sehen wir, dass dann ein Fehler verursacht wird:

```
l = [4,7,9,45,-3.7,99]
arithmetisches_Mittel(l)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-19288b78faae> in <module>
      1 l = [4,7,9,45,-3.7,99]
----> 2 arithmetisches_Mittel(l)

<ipython-input-9-4cb70ab2a6ea> in arithmetisches_Mittel(x, *l)
      6         sum += i
      7
----> 8     return sum / (1.0 + len(l))

TypeError: unsupported operand type(s) for /: 'list' and 'float'
```

Die Lösung besteht in der Benutzung eines weiteren Sternchens:

```
arithmetisches_Mittel(*l)
```

```
26.71666666666667
```

26 Python3 Namensraum

26.0.1 Namensräume und Geltungsbereiche

Namensräume

Allgemein gesprochen ist ein Namensraum (manchmal auch Kontext genannt) ein Benennungssystem, um Namen eindeutig zu machen und Mehrdeutigkeiten zu vermeiden. Ferner bezeichnet ein Namensraum einen Raum (oder Gebiet) innerhalb eines Programmes, in dem ein Variable gültig ist.

Namensräume kennt man nicht nur bei der Programmierung, sondern auch im Alltag:

- Der einfachste Falle eines Namensraumes stellt beispielsweise unser Namenssystem bei Personen dar. Im Freundeskreis langt es oft von einem "Peter", "Guido" oder einer "Susanne" zu sprechen. Bei Mehrdeutigkeiten greifen wir dann auf den Familiennamen zu. Das bedeutet, dass ein Familienname einen Namensraum bezeichnet. Im Namensraum "Schmidt" haben wir dann die diversen Vornamen.
- Ein weiteres Beispiel eines Namensraums finden wir bei der Telefonie. Fragt man beispielsweise einen Nachbarn nach seiner Telefonnummer, nennt dieser einem die Rufnummer ohne Vorwahl. Die Vorwahl stellt also einen Namensraum dar. Das bedeutet, dass eine Nummer wie z.B. die 3333 in der Stadt A mit der Vorwahl X einem Taxiunternehmen gehört und in der Stadt B mit der Vorwahl Y gehört diese Nummer möglicherweise einem Arzt. Die vollständigen Rufnummern also X/3333 und Y/3333 sind aber unterschiedlich.
- Netzwerke: Jedes Netzwerk-Gerät (Arbeitsplatz, Server, Drucker,…) braucht einen eindeutigen Namen und Addresse.
- Verzeichnisstrukturen in Dateisystemen liefern ein weiteres Beispiel. Der selbe Dateiname kann in verschiedenen Verzeichnissen verwendet werden. Der Zugriff ist nur über einen eindeutigen Pfad möglich.

Namensräume verhindern Namenskonflikte, d.h. dass man in verschiedenen Modulen zufällig gleiche Namen für Variablen, Funktionen usw. verwendet.

Viele Programmiersprachen verwenden Namensräume oder Geltunsbereiche für Bezeichner. Ein Bezeichner, der in einem Namensraum definiert ist, wird mit diesem Namensraum assoziiert. Auf diese Weise kann derselbe Bezeichner unabhängig voneinander in mehreren Namensräumen definiert werden (wie derselbe Dateiname in verschiedenen Verzeichnissen). Programmiersprachen, die Namensräume unterstützen, können unterschiedliche Regeln haben, die festlegen, zu welchem Namensraum ein Bezeichner gehört. Namensräume in Python sind als Python-Wörterbücher implementiert, d. h. sie werden durch eine Abbildung von Namen, d. h. den Schlüsseln des Wörterbuchs, auf Objekte, d. h. die Werte, definiert. Der Benutzer muss dies nicht wissen, um ein Python-Programm zu schreiben und wenn er Namensräume verwendet.

In Python gibt es drei Namensräume:

Lokaler Geltungsbereich Namen sind innerhalb einer Methode oder Funktion definiert.

Modularer Geltungsbereich Namen sind innerhalb einer Datei definiert.

Eingebauter Geltungsbereich Innerhalb von Python definierte Namen sind immer gültig.

Lebensdauer eines Namensraums

Nicht jeder Namensraum, der in einem Skript oder Programm verwendet werden kann, ist zu jedem Zeitpunkt während der Ausführung des Skripts zugänglich (oder lebendig). Namensräume haben unterschiedliche Geltungszeiten, da sie oft zu unterschiedlichen Zeitpunkten angelegt werden. Es gibt einen Namensraum, der von Anfang bis Ende vorhanden ist: Der Namensraum, der die eingebauten Namen enthält, wird beim Start des Python-Interpreters angelegt und wird nie gelöscht. Der globale Namensraum eines Moduls wird erzeugt, wenn das Modul eingelesen wird. Modul-Namensräume bestehen normalerweise bis zum Ende des Skripts, d. h. bis zum Beenden des Interpreters. Wenn eine Funktion aufgerufen wird, wird ein lokaler Namensraum für diese Funktion erzeugt. Dieser Namensraum wird entweder gelöscht, wenn die Funktion endet, d. h. zurückkehrt, oder wenn die Funktion eine Ausnahme auslöst, die nicht innerhalb der Funktion behandelt wird.

Geltungsbereiche

Ein Geltungsbereich bezeichnet einen Bereich des Programms, in dem auf einen Namensraum direkt zugegriffen werden kann, ohne einen Namens-Präfix zu verwenden. Zum Beispiel innerhalb einer Funktion. Mit anderen Worten, der Geltungsbereich eines Namens ist der Bereich eines Programms, in dem dieser Name eindeutig verwendet werden kann, z. B. innerhalb einer Funktion. Der Namensraum eines Namens ist identisch mit seinem Geltungsbereich. Geltungsbereiche sind statisch definiert, werden aber dynamisch verwendet. Während der Programmausführung gibt es folgende verschachtelte Geltungsbereiche:

- der innerste Geltungsbereich wird zuerst durchsucht und enthält die lokalen Namen
- die Geltungsbereiche aller umschließenden Funktionen, die beginnend mit dem nächstgelegenen umschließenden Geltungsbereich durchsucht werden
- der vorletzte Bereich enthält die globalen Namen des aktuellen Moduls
- der äußerste Bereich, der als letzter durchsucht wird, ist der Namensraum, der die eingebauten Namen enthält.

Der Namensraum eines Namens ist identisch zu dessen Geltungsbereich. Geltungsbereiche sind statisch, werden aber dynamisch verwendet.

27 Python3 Global Lokal

27.0.1 Globale und Lokale Variablen

Python behandelt globale und lokale Variablen in einer eigenwilligen Art. Während in vielen anderen Programmiersprachen Variablen automatisch global sind, wenn man sie nicht explizit als lokal deklariert, ist dies in Python genau anders herum. Die zugrundeliegende Idee besteht darin, dass die Benutzung von globalen Variabeln generell als schlechter Programmierstil betrachtet wird, weil dadurch viele Fehler und Seiteneffekte auftreten können. In den meisten Fällen, in denen man versucht ist, eine globale Variable zu verwenden, kann man den gewünschten Effekt besser mittels eines Funktionsparameters realisieren oder durch die Rückgabe eines Wertes mittels eines return-Wertes. Wie auch in vielen anderen Fällen, wird hier durch das Design von Python ein guter Programmierstil gewissermaßen erzwungen.

Das bedeutet, dass jede Variable, die man innerhalb einer Funktion definiert, automatisch einen lokalen Gültigkeitsbereich hat. Das bedeutet, dass was immer man mit dieser Variable innerhalb der Funktion macht, keinen Einfluss auf andere Variablen außerhalb der Funktion hat, auch wenn diese den gleichen Namen haben. Der Funktionsrumpf ist also der Gültigkeitsbereich einer solchen Variablen.

Um keine Missverständnisse aufkommen zu lassen: Variablen müssen nicht deklariert werden, wie dies in anderen Sprachen wie C und Java üblich ist. Variablen werden in Python implizit deklariert, wenn man sie definiert, d.h. ihnen einen Wert zuweist. Eine Variable erhält automatisch den richtigen Datentyp. Bei Problemen mit dieser Thematik empfehlen wir unser Kapitel über "Datentypen und Variablen", siehe Links auf der linken Seite.

Globale und lokale Variablen in Funktionen in Beispielen

Im folgenden Beispiel zeigen wir, wie globale Variablen innerhalb des Funktionsrumpfes benutzt werden können. Allerdings nur "lesend", also ohne den Wert zu ändern:

```
def f():
    print(s)
s = "I love Paris in the summer!"
f()
```

```
I love Paris in the summer!
```

Die Variable s wird definiert, in dem ihr die Zeichenkette "I love Paris in the summer!" zugeordnet wird. Diese Definition erfolgt vor dem Funktionsaufruf f(). Der Funktionsrumpf von f() besteht nur aus der "print(s)"-Anweisung. Weil es keine lokale Variable s gibt, d.h. keine Zuweisung an s innerhalb des Funktionsrumpfes von f, wird der Wert der globalen Variablen s benutzt. Dieser Wert kann natürlich nicht verändert werden, wie wir weiter unten in diesem Kapitel sehen werden. Es wird also der String "I love Paris in the summer!" ausgegeben.

Es stellt sich aber die Frage, was passiert, wenn wir den Wert von s innerhalb der Funktion von f() verändern. Wird dies eine Auswirkung auf die globale Variable s haben? Wir testen dies im folgenden kleinen Skript:

```
def f():
    s = "I love London!"
    print(s)

s = "I love Paris!"
f()
print(s)
```

```
I love London!
I love Paris!
```

Wie sieht es aber aus, wenn wir das erste Beispiel mit dem zweiten Beispiel kombinieren, d.h. wir also zuerst auf s mittels print zugreifen, in der Hoffnung den globalen Wert zu erhalten, und dann s einen neuen Wert zuweisen? Indem wir s einen Wert zuweisen können, machen wir s zu einer lokalen Variable. Dadurch gäbe es s innerhalb des Funktionsrumpfes sowohl als globale als auch als lokale Variable. Python lässt diese Mehrdeutigkeit nicht zu und es kommt zu einer Fehlermeldung, wie wir im folgenden Beispiel sehen können:

```
def f():
   print(s)
   s = "I love London!"
   print(s)

s = "I love Paris!"
f()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-3-d7a23bc83c27> in <module>
      5
      6 s = "I love Paris!"
----> 7 f()

<ipython-input-3-d7a23bc83c27> in f()
      1 def f():
----> 2    print(s)
      3    s = "I love London!"
      4    print(s)
      5

UnboundLocalError: local variable 's' referenced before assignment
```

Eine Variable kann nicht sowohl lokal als auch global innerhalb des gleichen Blocks, hier der Funktionsrumpf, sein. Deswegen betrachtete Python s als eine lokale Variable innerhalb des Rumpfes. Da nun auf diese lokale Variable zugegriffen wird, bevor sie definiert worden ist, sie also noch keinen Wert erhalten hat, erfolgt die Fehlermeldung.

Man kann jedoch auf globale Variablen “schreibend” innerhalb einer Funktion zugreifen. Dazu muss man sie jedoch explizit mittels des Schlüsselwortes “global” als global deklarieren. Wir können dies im folgenden Beispiel sehen:

```
def f():
    global s
    print(s)
    s = "Zur Zeit nicht, aber Berlin ist auch toll!"
    print(s)


s = "Gibt es einen Kurs in Paris?"
f()
print(s)
```

```
Gibt es einen Kurs in Paris?
Zur Zeit nicht, aber Berlin ist auch toll!
Zur Zeit nicht, aber Berlin ist auch toll!
```

Damit haben wir die Mehrdeutigkeit beseitigt.

Auf lokale Variablen einer Funktion kann von außen nicht zugegriffen werden:

```
def f():
    s = "I am globally not known"
    print(s)

f()
print(s)
```

```
I am globally not known
I love Paris in the summer!
```

Das folgende Beispiel zeigt eine wilde Kombination von lokalen und globalen Variablen und Funktionsparametern, um die obigen Sachverhalte nochmals per Beispiel zu vertiefen:

```
def foo(x, y):
    global a
    a = 42
    x,y = y,x
    b = 33
    b = 17
    c = 100
    print(a,b,x,y)

a,b,x,y = 1,15,3,4
foo(17,4)
print(a,b,x,y)
```

```
42 17 4 17
42 15 3 4
```

Globale Variablen in verschachtelten Funktionen

Wir werden jetzt untersuchen, was passieren wird, wenn wir das globale Schlüsselwort in verschachtelten Funktionen verwenden. Das folgende Beispiel zeigt eine Situation, in der eine Variable ‘Stadt’ in verschiedenen Bereichen verwendet wird:

```
def f ():
     Stadt = "Hamburg"
     def g ():
         global Stadt
         Stadt = "Genf"
     print ("Vor dem Aufruf von g:" + Stadt)
     print ("Jetzt anrufen g:")
     g()
     print ("Nach dem Aufruf von g:" + Stadt)

f ()
print ("Wert der Stadt in der Hauptsache:" + Stadt)
```

```
Vor dem Aufruf von g:Hamburg
Jetzt anrufen g:
Nach dem Aufruf von g:Hamburg
Wert der Stadt in der Hauptsache:Genf
```

Wir können sehen, dass die globale Anweisung innerhalb der verschachtelten Funktion g die Variable 'Stadt' der Funktion f nicht beeinflusst, d. H. Ihren Wert 'Hamburg' behält. Aus diesem Beispiel können wir auch ableiten, dass nach dem Aufruf von f () eine Variable 'Stadt' im Modul-Namespace existiert und den Wert 'Geneva' hat. Dies bedeutet, dass das globale Schlüsselwort in verschachtelten Funktionen keinen Einfluss auf den Namespace des umschließenden Namespace hat! Dies steht im Einklang mit dem, was wir im vorherigen Unterkapitel herausgefunden haben: Eine innerhalb einer Funktion definierte Variable ist lokal, sofern sie nicht explizit als global markiert ist. Mit anderen Worten, wir können auf einen Variablennamen in einem beliebigen umschließenden Bereich verweisen, aber wir können Variablennamen nur im lokalen Bereich neu binden, indem wir ihm zuweisen, oder im modulglobalen Bereich, indem wir eine globale Deklaration verwenden. Wir brauchen eine Möglichkeit, auch auf Variablen anderer Bereiche zuzugreifen. Der Weg dazu sind nichtlokale Definitionen, die wir im nächsten Kapitel erläutern werden.

nichtlokale Variablen

Python3 führte nichtlokale Variablen als neue Art von Variablen ein. Nichtlokale Variablen haben viel mit globalen Variablen gemeinsam. Ein Unterschied zu globalen Variablen besteht in der Tatsache, dass es nicht möglich ist, Variablen aus dem Modulbereich, d. H. Variablen, die nicht innerhalb einer Funktion definiert sind, mithilfe der nichtlokalen Anweisung zu ändern. Wir zeigen dies in den beiden folgenden Beispielen:

```
def f():
    global Stadt
    print(Stadt)

Stadt = "Frankfurt"
f()
```

```
Frankfurt
```

Dieses Programm ist korrekt und gibt 'Frankfurt' als Ausgabe zurück. Wir werden "global" in "nonlocal" im folgenden Programm ändern:

```
def f():
    nonlocal Stadt
    print(Stadt)

Stadt = "Frankfurt"
f()
```

```
  File "<ipython-input-11-5cb1f8add48e>", line 2
    nonlocal Stadt
    ^
SyntaxError: no binding for nonlocal 'Stadt' found
```

Dies zeigt, dass nichtlokale Bindungen nur innerhalb verschachtelter Funktionen verwendet werden können. Eine nichtlokale Variable muss im umschließenden Funktionsumfang definiert werden. Wenn die Variable nicht im umschließenden Funktionsbereich definiert ist, kann die Variable nicht im verschachtelten Bereich definiert werden. Dies ist ein weiterer Unterschied zur "globalen" Semantik.

```
def f ():
     Stadt = "München"
     def g ():
         nonlocal Stadt
         Stadt = "Zürich"
     print ("Vor dem Aufruf von g:" + Stadt)
     print ("Jetzt anrufen g:")
     g()
     print ("Nach dem Aufruf von g:" + Stadt)

Stadt = "Stuttgart"
f ()
print ("'Stadt' in der Hauptsache:" + Stadt)
```

```
Vor dem Aufruf von g:München
Jetzt anrufen g:
Nach dem Aufruf von g:Zürich
'Stadt' in der Hauptsache:Stuttgart
```

Im vorherigen Beispiel wurde die Variable 'Stadt' vor dem Aufruf von g definiert. Wir erhalten einen Fehler, wenn er nicht definiert ist:

```
def f ():
     #Stadt = "München"
     def g ():
         nonlocal Stadt
         Stadt = "Zürich"
     print ("Vor dem Aufruf von g:" + Stadt)
     print ("Jetzt anrufen g:")
     g()
     print ("Nach dem Aufruf von g:" + Stadt)

Stadt = "Stuttgart"
f ()
print ("'Stadt' in der Hauptsache:" + Stadt)
```

```
  Cell In[1], line 4
    nonlocal Stadt
    ^
SyntaxError: no binding for nonlocal 'Stadt' found
```

Das Programm funktioniert einwandfrei - mit oder ohne die Zeile ‘Stadt = “Munich”’ innerhalb von f -, wenn wir “nonlocal” in “global” ändern:

```
def f ():
    #Stadt = "München"
    def g ():
        global Stadt
        Stadt = "Zürich"
    print ("Vor dem Aufruf von g:" + Stadt)
    print ("Jetzt anrufen g:")
    g()
    print ("Nach dem Aufruf von g:" + Stadt)

Stadt = "Stuttgart"
f ()
print ("'Stadt' in der Hauptsache:" + Stadt)
```

```
Vor dem Aufruf von g:Stuttgart
Jetzt anrufen g:
Nach dem Aufruf von g:Zürich
'Stadt' in der Hauptsache:Zürich
```

Es gibt jedoch einen großen Unterschied: Der Wert des globalen x wird jetzt geändert!

28 Python3 Dateien

28.0.1 Dateien lesen und schreiben

Dateien

Dateien speichern Daten über die Laufzeit eines Programms hinaus und machen sie damit persistent. Python unterscheidet beim Öffnen grundsätzlich zwischen **Textmodus** und **Binärmodus**.

Zum Öffnen dient `open()`. Der Modus `rr"` (Text lesen) ist der Default. Weitere wichtige Modi sind `"w"` (schreiben und vorhandene Datei dabei leeren), `ä"` (anhängen), `*"` (nur neu anlegen) und `"b"` für Binärdaten.

Bei Textdateien sollte man die Zeichenkodierung bewusst angeben, beispielsweise `encoding=ütf-8"`, wenn das Dateiformat UTF-8 verwendet. Ohne Angabe ist die Standardkodierung plattformabhängig.

Dateien sollten zuverlässig geschlossen werden. Dafür ist ein `with`-Block normalerweise die beste Schreibweise, weil das Dateiobjekt auch bei einer Ausnahme automatisch geschlossen wird.

Sehr häufig verarbeitet man eine Textdatei zeilenweise. Dateiobjekte sind iterierbar; jede Iteration liefert eine Zeile. `rstrip()` kann den Zeilenumbruch für die Ausgabe entfernen:

```
with open("yellow_snow.txt", encoding="utf-8") as fobj:
    for line in fobj:
        print(line.rstrip())
```

Schreiben in eine Datei

Zum Schreiben verwendet man beispielsweise den Modus `"w"`. Achtung: Existiert die Datei bereits, wird ihr bisheriger Inhalt beim Öffnen im `w`-Modus verworfen. `write()` schreibt Strings in eine im Textmodus geöffnete Datei.

```
with open("yellow_snow.txt", encoding="utf-8") as fobj_in,
↪  open("yellow_snow2.txt", "w", encoding="utf-8") as fobj_out:
    for i, line in enumerate(fobj_in, start=1):
        print(line.rstrip())
        fobj_out.write(f"{i}: {line}")
```

Der `with`-Block verwendet das Dateiobjekt als Context Manager. Beim Verlassen des Blocks wird die Datei zuverlässig geschlossen – auch wenn innerhalb des Blocks eine Ausnahme auftritt. Ein manuelles `close()` ist dann nicht notwendig.

```
with open("example.txt", "w", encoding="utf-8") as fh:
```

```
    fh.write("To write or not to write\nthat is the question!\n")
```

Beim Modus `"w"` wird eine bereits vorhandene Datei beim Öffnen geleert. Soll vorhandener Inhalt erhalten bleiben und neuer Text am Ende ergänzt werden, verwendet man `ä"`. Soll das versehentliche Überschreiben einer existierenden Datei verhindert werden, ist `*"` nützlich.

Eine ganze Datei einlesen

Für kleine Dateien kann man den gesamten Inhalt auf einmal einlesen. `read()` liefert einen String, `readlines()` eine Liste von Zeilen. Bei großen Dateien ist die zeilenweise Iteration meist speicherschonender.

```
with open("ad_lesbiam.txt", encoding="utf-8") as fobj:
    gedicht = fobj.readlines()
print(gedicht)
```

```
print(gedicht[2])
```

```
VIVAMUS mea Lesbia, atque amemus,
```

Im obigen Beispiel wurde das Gedicht als Liste von Zeilen geladen. Die dritte Zeile steht entsprechend in `gedicht[2]`.

Mit `read()` erhält man stattdessen den gesamten Text als einen String:

```
with open("ad_lesbiam.txt", encoding="utf-8") as fobj:
    gedicht = fobj.read()
print(gedicht[16:34])
```

```
type(gedicht)
```

```
str
```

Persistente Daten

Persistente Daten stehen auch nach dem Ende eines Programmlaufs wieder zur Verfügung. Je nach Zweck eignen sich unterschiedliche Formate:

- **Text/JSON**: gut für interoperable und lesbare Datenstrukturen.
- **pickle**: Python-spezifische Serialisierung vieler Python-Objekte; nur für vertrauenswürdige Daten.
- **shelve**: persistente, Dictionary-ähnliche Ablage; verwendet intern `pickle`.
- **Datenbanken** wie SQLite: sinnvoll für strukturierte Daten, Abfragen und größere Datenbestände.

Das Modul `marshal` ist primär ein internes Format für Python und insbesondere für `.pyc`-Dateien; es ist kein allgemeiner Ersatz für `pickle`.

Pickle-Modul

Mit `pickle` lassen sich viele Python-Objektstrukturen in einen binären Datenstrom serialisieren und später wieder rekonstruieren. Typisch verwendet man `pickle.dump(obj, file)` zum Schreiben und `pickle.load(file)` zum Lesen.

Das Pickle-Protokoll hat sich über die Python-Versionen weiterentwickelt. Wird kein `protocol` angegeben, verwendet Python `pickle.DEFAULT_PROTOCOL`; dessen Wert kann sich zwischen Python-Versionen ändern. Einen festen Protokollwert sollte man daher vor allem dann angeben, wenn gezielt Kompatibilität mit älteren Python-Versionen benötigt wird.

Sicherheitshinweis: Pickle-Daten dürfen nur aus vertrauenswürdiger Quelle geladen werden. Beim Unpickling manipulierter Daten kann beliebiger Code ausgeführt werden. Für nicht vertrauenswürdige oder sprachübergreifend auszutauschende Daten ist beispielsweise JSON meist die bessere Wahl.

Ein einfaches Beispiel:

```
import pickle

staedte = ["Bern", "Basel", "St. Gallen", "Zürich"]
with open("data.pkl", "wb") as fh:
    pickle.dump(staedte, fh)
```

`pickle.load()` erkennt das im Datenstrom verwendete Pickle-Protokoll automatisch. Die Datei wird im Binärmodus geöffnet:

```
import pickle

with open("data.pkl", "rb") as fh:
    staedte = pickle.load(fh)
print(staedte)
```

Da `pickle` Objekte und ihre Struktur serialisiert, nicht die Variablennamen des aufrufenden Programms, weist man das Ergebnis von `pickle.load()` wieder einem Namen zu.

Pickle unterstützt viele eingebaute Typen, verschachtelte Container und zahlreiche benutzerdefinierte Klasseninstanzen. Funktionen und Klassen werden im Allgemeinen über ihren qualifizierten Namen referenziert; die entsprechenden Definitionen müssen beim Laden importierbar sein. Nicht jedes Python-Objekt ist pickelbar – geöffnete Dateiobjekte sind beispielsweise ein typischer Gegenfall.

Mehrere zusammengehörige Objekte kann man in einem Container wie einem Tupel, einer Liste oder einem Dictionary gemeinsam serialisieren:

```
import pickle

programmiersprachen = ["Python", "Perl", "C++", "Java", "Lisp"]
python_dialekte = ["Jython", "IronPython", "CPython"]
pickle_object = (programmiersprachen, python_dialekte)
with open("data.pkl", "wb") as fh:
    pickle.dump(pickle_object, fh)
```

Obige gepickelte Datei können wir so einlesen, dass wir die beiden Listen wieder trennen können:

```
import pickle

with open("data.pkl", "rb") as fh:
    sprachen, dialekte = pickle.load(fh)
print(sprachen, dialekte)
```

`shelve`-Modul

Ein Shelf ist eine persistente, Dictionary-ähnliche Ablage. Die Schlüssel sind Strings; als Werte sind Objekte möglich, die `pickle` serialisieren kann.

Weil `shelve` intern `pickle` verwendet, gilt derselbe Sicherheitshinweis: Ein Shelf aus nicht vertrauenswürdiger Quelle darf nicht geöffnet werden. Außerdem sollte ein Shelf zuverlässig geschlossen werden. `shelve.open()` kann dafür direkt als Context Manager verwendet werden.

Ein wichtiger Unterschied zu einem normalen Dictionary betrifft veränderliche Werte: Ohne `writeback=True` werden Änderungen an einem ausgelesenen veränderlichen Objekt nicht automatisch zurückgeschrieben. Häufig ist es klarer, den Wert zu lesen, zu verändern und anschließend explizit wieder unter demselben Schlüssel zu speichern.

```
import shelve

with shelve.open("MyShelve") as s:
    s["strasse"] = "Fleet Str"
    s["stadt"] = "London"
    for key in s:
        print(key)
```

Beim nächsten Öffnen stehen die gespeicherten Werte wieder zur Verfügung:

```
import shelve

with shelve.open("MyShelve") as s:
    print(s["strasse"])
    print(s["stadt"])
    print(dict(s))
```

Auch komplexere, pickelbare Objekte können als Werte gespeichert werden. Bei verschachtelten veränderlichen Objekten sollte man Änderungen explizit zurückschreiben:

```
import shelve

with shelve.open("MyPhoneBook") as tele:
    tele["Mike"] = {"Vorname": "Mike", "Nachname": "Miller", "Telefon": "4689"}
    tele["Steve"] = {"Vorname": "Stephan", "Nachname": "Burns", "Telefon":
    ↪  "8745"}
    tele["Eve"] = {"Vorname": "Eve", "Nachname": "Naomi", "Telefon": "9069"}

    eve = tele["Eve"]
```

```
    eve["Telefon"] = "9070"
    tele["Eve"] = eve
```

Die explizite erneute Zuweisung `tele[Ëve"] = eve` sorgt dafür, dass die Änderung persistiert. Alternativ kann ein Shelf mit `writeback=True` geöffnet werden; dies ist komfortabler, kann bei vielen gelesenen Einträgen aber deutlich mehr Speicher verbrauchen und das Schließen verlangsamen.

```
with shelve.open("MyPhoneBook") as tele:
    print(tele["Eve"]["Telefon"])
```

Moderne Alternative: `pathlib`

Für einfache Dateioperationen bietet `pathlib.Path` eine objektorientierte Schnittstelle. `Path.read_text()` und `Path.write_text()` öffnen und schließen Textdateien automatisch:

```
from pathlib import Path

pfad = Path("notiz.txt")
pfad.write_text("Hallo Python!\n", encoding="utf-8")
text = pfad.read_text(encoding="utf-8")
```

Übungen

Die Datei cities_and_times.txt enthält Städtenamen und Zeiten. Jede Zeile enthält als ersten Eintrag den Namen einer Stadt, gefolgt von einem Wochentag und einer Zeitangabe in der Form hh:mm. Lesen Sie die Datei ein und erzeugen Sie eine alphabetisch geordnete Liste der Form

```
[('Amsterdam', 'Sun', (8, 52)), ('Anchorage', 'Sat', (23, 52)), ('Ankara', 'Sun', (10, 52))

...

 ('Toronto', 'Sun', (2, 52)), ('Vancouver', 'Sun', (0, 52)), ('Vienna', 'Sun', (8, 52)), ('
```

Anschließend soll die Liste zur späteren Benutzung mit Hilfe des `pickle`-Moduls abgespeichert werden. Wir verwenden diese Datei später im NumPy-Kapitel über Datentypen.

Lösung

```
import pickle

with open("cities_and_times.txt", encoding="utf-8") as fh:
    lines = sorted(fh)

cities = []
for line in lines:
    *city, day, time = line.split()
```

```
    hours, minutes = time.split(":")
    cities.append((" ".join(city), day, (int(hours), int(minutes))))

with open("cities_and_times.pkl", "wb") as fh:
    pickle.dump(cities, fh)
```

Städtenamen können mehrere Wörter enthalten. Deshalb wird beim Entpacken der Zeile `*city` verwendet: Alle Wörter vor Wochentag und Uhrzeit landen in einer Liste. `". join(city)` setzt diese Wörter wieder zu einem Städtenamen zusammen, zum Beispiel `SSalt Lake City"`.

29 Python3 Modularisierung

29.0.1 Modulare Programmierung und Module

Modulare Programmierung

Modulare Programmierung zerlegt größere Programme in überschaubare, möglichst klar abgegrenzte Komponenten. In Python ist ein Modul typischerweise eine `.py`-Datei mit Definitionen und ausführbaren Anweisungen. Module schaffen eigene Namensräume und erleichtern Wiederverwendung, Tests und Wartung.

```
import math
```

Das Modul math stellt mathematische Konstanten und Funktionen zur Verfügung, z. B. (math.pi), die Sinusfunktion (math.sin()) und die Cosinusfunktion (math.cos()). Auf jedes Attribut bzw. jede Funktion kann nur durch Voranstellen von "math." zugegriffen werden:

```
math.pi
```

```
3.141592653589793
```

```
math.sin(math.pi/2)
```

```
1.0
```

```
math.cos(math.pi/2)
```

```
6.123233995736766e-17
```

```
math.cos(math.pi)
```

```
-1.0
```

Mehrere Module können syntaktisch in einer `import`-Anweisung genannt werden. Für bessere Lesbarkeit empfiehlt es sich in normalem Quellcode jedoch meist, pro Zeile ein Modul zu importieren:

```
import math
import random
```

Import-Anweisungen können an beliebiger Stelle im Programm platziert werden, aber es gehört zum guten Stil, sie direkt am Anfang eines Programms zu platzieren.

Wenn nur bestimmte Objekte eines Moduls benötigt werden, können wir nur diese importieren:

```
from math import sin, pi
```

Die anderen Objekte, z. B. cos, sind nach diesem Import nicht mehr verfügbar. Wir sind in der Lage, auf sin und pi direkt zuzugreifen, d. h. ohne ihnen das Präfix "math" voranzustellen. Anstatt bestimmte Objekte aus einem Modul explizit zu importieren, ist es auch möglich, alles im Namensraum des importierenden Moduls zu importieren. Dies kann durch die Verwendung eines Sternchens im Import erreicht werden:

```
from math import *
sin(3.01) + tan(cos(2.1)) + e
```

```
2.2968833711382604
```

```
e
```

```
2.718281828459045
```

Die Sternchen-Schreibweise (`from modul import *`) sollte in normalem Programmcode vermieden werden. Sie macht schwer nachvollziehbar, woher ein Name stammt, und kann bestehende Namen unbemerkt überschreiben. Explizite Modulnamen oder gezielte Importe sind deutlich robuster.

```
from numpy import *
from math import *
print(sin(3))
```

```
0.1411200080598672
```

```
sin(3)
```

```
0.1411200080598672
```

Lassen Sie uns das vorherige Beispiel leicht modifizieren, indem wir die Reihenfolge der Importe ändern:

```
from math import *
from numpy import *
print(sin(3))
```

```
0.1411200080598672
```

```
sin(3)
```

```
0.1411200080598672
```

Sternimporte wirken bequem, weil Namen ohne Modulpräfix verwendet werden können. In größerem Code wird dadurch aber die Herkunft von Namen unklar. Ein verbreiteter und besser lesbarer Weg, Tipparbeit zu sparen, sind **Modul-Aliase**. Bei NumPy ist beispielsweise folgende Konvention üblich:

```
import numpy as np
```

Damit bleibt der Namensraum sichtbar und NumPy-Objekte werden mit `np.` angesprochen:

```
import numpy as np
np.diag([3, 11, 7, 9])
```

```
array([[ 3,  0,  0,  0],
       [ 0, 11,  0,  0],
       [ 0,  0,  7,  0],
       [ 0,  0,  0,  9]])
```

Module entwerfen und schreiben

Ein Python-Modul ist im einfachsten Fall eine Datei mit der Endung `.py`, die Definitionen und ausführbare Anweisungen enthält. Der Modulname entspricht normalerweise dem Dateinamen ohne `.py`.

Als Beispiel speichern wir zwei Fibonacci-Funktionen in `fibonacci.py`:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)


def ifib(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a
```

Liegt `fibonacci.py` auf dem Modul-Suchpfad, kann es importiert und über seinen Modul-Namensraum verwendet werden:

```
import fibonacci
fibonacci.fib(7)
```

```
13
```

```
fibonacci.fib(20)
```

```
6765
```

```
fibonacci.ifib(42)
```

```
267914296
```

```
fibonacci.ifib(1000)
```

```
434665576869374564356885276750406258025646605173717804024817290895365554179490518904038798400792551692959225930803226347752096896232398733224711616429964409065331879382989696499285160037044761377951668492288751
```

Die rekursive Variante ist für große Argumente absichtlich ungeeignet: Sie berechnet dieselben Teilprobleme immer wieder. Die iterative Variante ist für dieses Beispiel wesentlich effizienter.

Wer einen häufig verwendeten Namen abkürzen möchte, kann ihn lokal binden; meist sind aber ein expliziter Import oder der Modul-Namensraum lesbarer:

```
fib = fibonacci.ifib
fib(10)
```

```
55
```

Besser ist es in vielen Fällen, die benötigten Namen gezielt zu importieren oder – besonders bei großen Bibliotheken – den Modul-Namensraum ausdrücklich beizubehalten.

Mehr über Module

Ein Modul darf neben Funktions- und Klassendefinitionen auch ausführbare Top-Level-Anweisungen enthalten. Beim ersten erfolgreichen Import in einer Interpreter-Sitzung wird der Modulcode ausgeführt und das Modul anschließend in `sys.modules` zwischengespeichert. Weitere normale Imports desselben Modulnamens verwenden dieses bereits geladene Modul.

```
print("The module is imported now!")
```

```
The module is imported now!
```

Wir speichern diese eine Anweisung unter dem Namen `one_time.py` und importieren das Modul:

```
import one_time
```

```
The module is imported now!
```

Ein normaler zweiter `import one_time` führt den Top-Level-Code nicht erneut aus, solange dasselbe Modul bereits in `sys.modules` geladen ist.

Bei interaktiver Entwicklung kann `importlib.reload()` ein bereits importiertes Modul erneut ausführen:

```
from importlib import reload
import one_time
reload(one_time)
```

`reload()` ist vor allem ein Werkzeug für interaktive Entwicklung. Bereits mit `from modul import name` gebundene Namen und schon erzeugte Instanzen werden dadurch nicht auto-

matisch auf neue Definitionen umgebogen. In regulären Programmen sollte die Logik daher nicht von wiederholtem Reloading abhängen.

Jedes Modul besitzt einen eigenen globalen Namensraum. Funktionen, die im Modul definiert sind, verwenden diesen Modul-Namensraum für globale Namen. Auf öffentliche Modulattribute greift man mit Punktnotation wie `modul.name` zu. Ein Modul kann seinerseits weitere Module importieren; Importe stehen nach üblicher Stilkonvention am Anfang einer Datei.

Direktes Importieren von Namen aus einem Modul

Namen können auch direkt in den Namensraum des importierenden Moduls gebunden werden:

```
from fibonacci import fib, ifib
ifib(500)
```

```
139423224561697880139724382870407283950070256587697307264108962948325571622863296915576588762225212941250
```

Dabei wird der Name `fibonacci` durch diese `from`-Anweisung nicht zusätzlich gebunden. Ein Sternimport (`from fibonacci import *`) importiert grundsätzlich die dafür vorgesehenen öffentlichen Namen, macht deren Herkunft aber schwer nachvollziehbar und sollte in normalem Programmcode vermieden werden:

```
from fibonacci import *
fib(500)
```

Module direkt als Skript ausführen

Eine Moduldatei kann auch direkt gestartet werden, zum Beispiel:

```
python fibonacci.py 10
```

Wird eine Datei direkt ausgeführt, hat die eingebaute Modulvariable `__name__` den Wert `"__main__"`. Beim Import als Modul enthält `__name__` dagegen den Modulnamen. Damit lässt sich Code formulieren, der nur beim direkten Start ausgeführt wird:

```
if __name__ == "__main__":
    import sys
    print(fib(int(sys.argv[1])))
```

Ein direkter Aufruf kann dann beispielsweise liefern:

```
$ python fibonacci.py 10
55
```

Beim normalen `import fibonacci` wird dieser `if`-Block nicht ausgeführt. Die Funktionen und anderen Modulobjekte stehen anschließend über den importierten Modul-Namensraum zur Verfügung:

```
import fibonacci
print(fibonacci.ifib(10))
```

Modulnamen mit `as` abkürzen

Mit `as` bindet man das importierte Modul unter einem lokalen Alias:

```
import math as mathematics
print(mathematics.cos(mathematics.pi))
```

```
-1.0
```

Die Anweisung `import math as mathematics` bindet das Modul unter dem Namen `mathematics`. Sie erzeugt durch diese Importanweisung **keine zusätzliche Bindung namens `math`**. Ob `math` trotzdem existiert, hängt davon ab, ob dieser Name vorher bereits im aktuellen Namensraum gebunden wurde.

Auch einzelne importierte Namen können mit `as` umbenannt werden.

```
from math import pi, pow as power, sin as sinus
power(2, 3)
```

```
8.0
```

```
sinus(pi)
```

```
1.2246467991473532e-16
```

Arten von Modulen

Es gibt verschiedene Arten von Modulen:

- Solche, die in Python geschrieben sind Sie haben die Endung: .py
- Dynamisch gelinkte C-Module Die Suffixe sind: .dll, .pyd, .so, .sl, ...
- Mit dem Interpreter gelinkte C-Module Es ist möglich, eine vollständige Liste dieser Module zu erhalten:

```
import sys
print(sys.builtin_module_names)
```

```
('_abc', '_ast', '_bisect', '_blake2', '_codecs', '_codecs_cn', '_codecs_hk',
↪  '_codecs_iso2022', '_codecs_jp', '_codecs_kr', '_codecs_tw', '_collections',
↪  '_contextvars', '_csv', '_datetime', '_functools', '_heapq', '_imp', '_io',
↪  '_json', '_locale', '_lsprof', '_md5', '_multibytecodec', '_opcode',
↪  '_operator', '_pickle', '_random', '_sha1', '_sha256', '_sha3', '_sha512',
↪  '_signal', '_sre', '_stat', '_string', '_struct', '_symtable', '_thread',
↪  '_tracemalloc', '_warnings', '_weakref', '_winapi', 'array', 'atexit',
↪  'audioop', 'binascii', 'builtins', 'cmath', 'errno', 'faulthandler', 'gc',
↪  'itertools', 'marshal', 'math', 'mmap', 'msvcrt', 'nt', 'parser', 'sys',
↪  'time', 'winreg', 'xxsubtype', 'zipimport', 'zlib')
```

Für Built-in-Module wird eine Fehlermeldung zurückgegeben.

Suchpfad für Module

Beim Import sucht Python über den Modul-Suchpfad `sys.path`. Dessen genaue Initialisierung hängt davon ab, wie Python gestartet wurde. Typischerweise enthält er zunächst das Verzeichnis des ausgeführten Skripts beziehungsweise bei einer interaktiven Sitzung das aktuelle Verzeichnis, danach unter anderem Einträge aus `PYTHONPATH` und installationsabhängige Standardverzeichnisse.

Den tatsächlich verwendeten Suchpfad sollte man deshalb nicht über fest verdrahtete Verzeichnisnamen erklären, sondern direkt abfragen:

```
import sys
for path in sys.path:
    print(path)
```

Nach einem Import kann `module.__file__` bei dateibasierten Modulen anzeigen, aus welcher Datei ein Modul geladen wurde:

```
import random
print(random.__file__)
```

Bei eingebauten oder anders bereitgestellten Modulen muss `__file__` nicht auf eine normale Quelldatei verweisen beziehungsweise kann fehlen.

```
import math
math.__file__
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-12-bb98ec32d2a8> in <module>
      1 import math
----> 2 math.__file__

AttributeError: module 'math' has no attribute '__file__'
```

Inhalt eines Moduls

Mit der eingebauten Funktion dir() und dem Namen des Moduls als Argument, können Sie alle gültigen Attribute und Methoden für dieses Modul auflisten.

```
import math
dir(math)
```

```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
```

```
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
 'erfc',
 'exp',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'ldexp',
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'pi',
 'pow',
 'radians',
 'remainder',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'tau',
 'trunc']
```

Wenn Sie dir() ohne Argument aufrufen, wird eine Liste mit den Namen im aktuellen lokalen Bereich zurückgegeben:

```
import math
cities = ["New York", "Toronto", "Berlin", "Washington", "Amsterdam",
↪  "Hamburg"]
dir()
```

```
['In',
 'Out',
```

```
 '_',
 '_1',
 '__',
 '___',
 '__builtin__',
 '__builtins__',
 '__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_dh',
 '_i',
 '_i1',
 '_i2',
 '_ih',
 '_ii',
 '_iii',
 '_oh',
 'builtins',
 'cities',
 'exit',
 'get_ipython',
 'math',
 'quit']
```

Es ist möglich, eine Liste der eingebauten Funktionen, Exceptions und anderer Objekte zu erhalten, indem Sie das builtins-Modul importieren:

```
import builtins
dir(builtins)
```

```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError',
 'ConnectionError',
 'ConnectionRefusedError',
 'ConnectionResetError',
 'DeprecationWarning',
 'EOFError',
 'Ellipsis',
 'EnvironmentError',
 'Exception',
 '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',
'WindowsError',
'ZeroDivisionError',
'__IPYTHON__',
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
```

```
 'ascii',
 'bin',
 'bool',
 'breakpoint',
 'bytearray',
 'bytes',
 'callable',
 'chr',
 'classmethod',
 'compile',
 'complex',
 'copyright',
 'credits',
 'delattr',
 'dict',
 'dir',
 'display',
 'divmod',
 'enumerate',
 'eval',
 'exec',
 'filter',
 'float',
 'format',
 'frozenset',
 'get_ipython',
 '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',
 'range',
 'repr',
 'reversed',
 'round',
```

```
 'set',
 'setattr',
 'slice',
 'sorted',
 'staticmethod',
 'str',
 'sum',
 'super',
 'tuple',
 'type',
 'vars',
 'zip']
```

Pakete

Wenn Sie irgendwann einmal viele Module erstellt haben, verlieren Sie vielleicht den Überblick über diese. Sie haben vielleicht Dutzende oder Hunderte von Modulen und diese können in verschiedene Kategorien eingeteilt werden. Es ist vergleichbar mit der Situation in einem Dateisystem: Anstatt alle Dateien in einem einzigen Verzeichnis zu haben, legt man sie in verschiedenen Verzeichnissen ab, die nach den Themen der Dateien geordnet sind. Wir werden im nächsten Kapitel unseres Python-Tutorials zeigen, wie man Module in Paketen organisiert.

30 Python3 Pakete

30.0.1 Pakete in Python

Einführung

Pakete strukturieren den Python-Modulnamensraum mit gepunkteten Namen wie `A.B`. Ein **reguläres Paket** ist typischerweise ein Verzeichnis mit Modulen und einer Datei `__init__.py`. Diese Datei kann leer sein oder Initialisierungscode und die öffentliche Paketoberfläche definieren.

Daneben gibt es **Namespace-Packages**, die ohne `__init__.py` auskommen und sich sogar über mehrere Verzeichnisse verteilen können. Für die grundlegenden Beispiele dieses Kapitels verwenden wir reguläre Pakete, weil sich daran Importverhalten und relative Importe besonders anschaulich zeigen lassen.

```
def bar():
    print("Hello, function 'bar' from module 'a' calling")
```

Der Inhalt von b.py:

```
def foo():
    print("Hello, function 'foo' from module 'b' calling")
```

Für unser **reguläres** Beispiel-Package legen wir außerdem eine leere Datei `__init__.py` im Verzeichnis `simple_package` an.

Schauen wir uns an, was passiert, wenn wir simple_package aus der interaktiven Python-Shell importieren, unter der Annahme, dass sich das Verzeichnis simple_package entweder in dem Verzeichnis befindet, aus dem Sie die Shell aufrufen, oder dass es im Suchpfad oder der Umgebungsvariablen “PYTHONPATH” (Ihres Betriebssystems) enthalten ist:

```
import simple_package
```

```
print(hasattr(simple_package, "a"))
```

```
print(hasattr(simple_package, "b"))
```

Der Import des Pakets allein lädt die Untermodule `a` und `b` nicht automatisch. Wir können sie explizit importieren:

```
from simple_package import a, b
a.bar()
b.foo()
```

```
Hello, function 'bar' from module 'a' calling
Hello, function 'foo' from module 'b' calling
```

Wie wir zu Beginn des Kapitels gesehen haben, können wir weder auf “a” noch auf “b” zugreifen, indem wir lediglich simple_package importieren.

Es gibt jedoch eine Möglichkeit, diese Module automatisch zu laden. Dazu können wir die Datei `__init__.py` verwenden. Alles, was wir tun müssen, ist, die folgenden Zeilen in die bisher leere Datei `__init__.py` einzufügen:

Es wird jetzt funktionieren:

```
import simple_package
simple_package.a.bar()
simple_package.b.foo()
```

```
Hello, function 'bar' from module 'a' calling
Hello, function 'foo' from module 'b' calling
```

Ein komplexeres Paket

Im folgenden Beispiel wollen wir demonstrieren, wie wir ein komplexeres Paket erstellen können. Wir werden das hypothetische Sound-Modul verwenden, das im offiziellen Tutorial verwendet wird. (siehe https://docs.python.org/3/tutorial/modules.html)

Wir werden eine Dummy-Implementierung - nur leere Dateien mit den richtigen Namen - dieser Struktur implementieren. Wir werden verschiedene Variationen der Implementierungen bereitstellen. Um zwischen den Implementierungen zu unterscheiden, werden wir die Module sound1, sound2, sound3 und sound4 nennen. Im Grunde sollten sie alle `Sound` heißen. Sie können die Beispiele als bzip-Dateien herunterladen:

- sounds1.tar.bz2
- sounds2.tar.bz2
- sounds3.tar.bz2
- sounds4.tar.bz2
- sounds5.tar.bz2
- sounds6.tar.bz2
- sounds7.tar.bz2

Wir werden mit dem Paket `Sound1` beginnen. (Sie können es mit ‘`tar xvjf sound1.tar.bz2` entpacken)

Wenn wir das Paket `sound1`` mit der Anweisung`import sound1`importieren, wird das Paket ```sound1` importiert, nicht aber die Unterpakete `effects``,filtersund ```formats`, wie wir im folgenden Beispiel sehen werden. Der Grund dafür liegt darin, dass die Datei `__init__.py` keinen Code für den Import von Unterpaketen enthält:

```
import sound1

print(sound1)

print(sound1.effects)
```

```
<module 'sound1' from '/data/Dropbox (Bodenseo)/Bodenseo Team
↪  Folder/melisa/notebooks_en/sound1/__init__.py'>

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-2-0b6d7fed3b24> in <module>
      3 print(sound1)
      4
----> 5 print(sound1.effects)

AttributeError: module 'sound1' has no attribute 'effects'
```

Wenn Sie auch das Unterpaket `effects` verwenden wollen, müssen Sie es explizit importieren, zum Beispiel mit `import sound1.effects`:

```
import sound1.effects
print(sound1.effects)
```

```
<module 'sound1.effects' from '/data/Dropbox (Bodenseo)/Bodenseo Team
↪  Folder/melisa/notebooks_en/sound1/effects/__init__.py'>
```

Es ist möglich, den Import der Submodule automatisch beim Import des Moduls `Sound1` durchführen zu lassen. Wir werden nun zu `Sound2` wechseln, um zu demonstrieren, wie man das macht. Wir verwenden die gleichen Dateien wie in `Sound1`, aber wir fügen die Codezeile `import sound2.effects` in die Datei `__init__.py`` des Verzeichnisses ``Sound2` ein. Die Datei sollte jetzt wie folgt aussehen:

```
"""Initialisierung des Pakets sound2."""
import sound2.effects

print("sound2.effects wird importiert!")
```

Wenn wir das Paket `sound2` aus der interaktiven Python-Shell importieren, sehen wir, dass auch das Unterpaket `effects` automatisch geladen wird:

```
import sound2
```

```
sound2 package is getting imported!
```

Anstatt einen absoluten Pfad zu verwenden, hätten wir das Effekt-Paket auch relativ zum Sound-Paket importieren können.

Wir werden dies in dem Modul `Sound3` demonstrieren:

```
import sound3
```

```
effects package is getting imported!
```

Es ist auch möglich, Module automatisch zu importieren, wenn das Unterpaket `effects` importiert wird. Dazu können relative Importe in der `__init__.py`-Datei des Verzeichnisses `effects` verwendet werden:

Durch den Import von `Sound4` werden automatisch auch die Module `Formate` und `Effekte` importiert:

```
import sound4
```

Zum Abschluss dieses Unterkapitels wollen wir zeigen, wie wir das Modul karaoke aus dem Paket filters importieren, wenn wir das Paket effects importieren. Zu diesem Zweck fügen wir die Zeile from ..filters import karaoke in die Datei `__init__.py` des Verzeichnisses effects ein. Die komplette Datei sieht nun wie folgt aus:

Das Importieren von `sound4` führt zu folgender Ausgabe:

```

import sound5
```

```
formats package is getting imported!
importing from the effects package:
formats module is getting imported!
filters package is getting imported!
Module karaoke.py has been loaded!
karaoke module is getting imported!
effects package is getting imported!
```

Wir können jetzt auf die Funktionen von Karaoke zugreifen und diese nutzen:

```

sound5.filters.karaoke.func1()
```

```
Funktion func1 has been called!
```

`from paket import *`

Ein Sternimport aus einem Paket importiert **nicht automatisch alle Dateien und Unterpakete des Verzeichnisses**. Welche Namen exportiert werden, hängt insbesondere von `__all__` und von den Namen ab, die beim Initialisieren des Pakets bereits gebunden wurden. Sternimporte sollten in Anwendungscode trotzdem möglichst vermieden werden.

```

from sound6 import *
```

```
sound package is getting imported!
```

Wir erhalten also die beruhigende Meldung, dass das Sound-Paket importiert wurde. Wenn wir jedoch mit der Funktion dir prüfen, sehen wir, dass weder das Modul foobar noch die Unterpakete effects, filters und formats importiert wurden:

```
for mod in ['foobar', 'effects', 'filters', 'formats']:
    print(mod, mod in dir())
```

```
foobar False
effects False
filters False
```

```
formats False
```

Mit `__all__` kann ein Paket explizit festlegen, welche Namen bei `from paket import *` exportiert werden sollen. Die Einträge sind Strings und müssen als exportierbare Namen des Pakets verfügbar gemacht werden. `__all__` dient damit als explizite Beschreibung dieser Sternimport-Oberfläche; es ersetzt nicht die generell besser lesbaren expliziten Importe.

Wir fügen nun die Zeile

in die Datei `__init__.py` des Soundverzeichnisses ein. Wir erhalten nun ein völlig anderes Ergebnis:

```
from sound7 import *
```

```
sound package is getting imported!
formats package is getting imported!
filters package is getting imported!
effects package is getting imported!
foobar module is getting imported
```

Auch wenn es bereits ersichtlich ist, dass alle Module importiert wurden, können wir mit dir noch einmal nachsehen:

```
for mod in ['foobar', 'effects', 'filters', 'formats']:
    print(mod, mod in dir())
```

```
foobar True
effects True
filters True
formats True
```

Die nächste Frage ist, was importiert wird, wenn wir * in einem Subpackage verwenden:

Wie erwartet sind die Module innerhalb von effects nicht automatisch importiert worden. Also können wir die folgende `__all__`-Liste in die `__init__.py`-Datei des Pakets effects einfügen:

Für die Unterpakete könnten beispielsweise folgende `__all__`-Listen verwendet werden:

```
# filters/__init__.py
__all__ = ["equalizer", "karaoke", "vocoder"]

# formats/__init__.py
__all__ = ["aiffread", "aiffwrite", "auread", "auwrite", "wavread", "wavwrite"]

# effects/__init__.py
__all__ = ["echo", "surround", "reverse"]
```

`__init__` selbst gehört normalerweise nicht in diese Liste.

Jetzt erhalten wir das beabsichtigte Ergebnis:

```
from sound8 import *
```

```
sound package is getting imported!
formats package is getting imported!
filters package is getting imported!
effects package is getting imported!
foobar module is getting imported
```

```
from sound8.effects import *
```

```
Module echo.py has been loaded!
Module surround.py has been loaded!
Module reverse.py has been loaded!
```

```
from sound8.filters import *
```

```
Module equalizer.py has been loaded!
Module karaoke.py has been loaded!
Module vocoder.py has been loaded!
```

```
from sound8.formats import *
```

```
Module aiffread.py has been loaded!
Module aiffwrite.py has been loaded!
Module auread.py has been loaded!
Module auwrite.py has been loaded!
Module wavread.py has been loaded!
Module wavwrite.py has been loaded!
```

Sternimporte erschweren statische Analyse, Autovervollständigung und die Nachvollziehbarkeit von Namen. In normalem Programmcode sind explizite Importe daher vorzuziehen, beispielsweise `from sound.effects import echo` oder `import sound.effects.echo`.

31 Python3 Ausnahmebehandlung

31.0.1 Fehler und Ausnahmen

Ausnahmebehandlung

Python unterscheidet unter anderem Syntaxfehler und **Ausnahmen** (Exceptions), die während der Ausführung auftreten. Eine Ausnahme ist zunächst ein Signal dafür, dass der normale Programmfluss nicht fortgesetzt werden kann. Viele Ausnahmen repräsentieren Fehlerzustände – beispielsweise ungültige Eingaben oder fehlende Dateien –, einige dienen aber auch speziellen Steuerungszwecken.

Mit `try` und `except` kann ein Programm erwartbare Ausnahmesituationen gezielt behandeln. Dabei sollte man möglichst **spezifische Exception-Klassen** abfangen. Ein nacktes `except:` fängt auch Systemausnahmen wie `KeyboardInterrupt` und `SystemExit`; in normalem Anwendungscode ist `except Exception as exc:` meist die bessere allgemeine Grenze.

Eine Ausnahme, für die kein passender Handler gefunden wird, propagiert durch die Aufrufkette weiter und beendet schließlich den aktuellen Programmablauf mit einem Traceback.

```
n = int(input("Bitte gebe eine Zahl ein: "))
```

```
Bitte gebe eine Zahl ein:  vier

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-b596fc0d106f> in <module>
----> 1 n = int(input("Bitte gebe eine Zahl ein: "))

ValueError: invalid literal for int() with base 10: 'vier'
```

Mit Hilfe der Ausnahmebehandlung können wir robusten Code für das Lesen eines Integers von der Eingabe schreiben:

```
while True:
    try:
        n = input("Bitte gebe eine ganze Zahl ein:  ")
        n = int(n)
        break
    except ValueError:
        print("Keine gültige Zahl, probiere es nochmals ...")
print("Großartig! Das war eine korrekte ganze Zahl")
```

```
Bitte gebe eine ganze Zahl ein:   vier

Keine gültige Zahl, probiere es nochmals ...

Bitte gebe eine ganze Zahl ein:   5.6
```

```
Keine gültige Zahl, probiere es nochmals ...

Bitte gebe eine ganze Zahl ein:   5

Großartig! Das war eine korrekte ganze Zahl
```

Es handelt sich um eine Schleife, die nur dann abbricht, wenn eine gültige Ganzzahl angegeben wurde. Die while-Schleife wird eingegeben. Der Code innerhalb der try-Klausel wird Anweisung für Anweisung ausgeführt. Tritt während der Ausführung keine Exception auf, wird die Ausführung bis zur break-Anweisung geführt und die while-Schleife verlassen. Tritt eine Exception auf, z. B. beim Casting von n, wird der Rest des try-Blocks übersprungen und die except-Klausel ausgeführt. Der ausgelöste Fehler, in unserem Fall ein ValueError, muss mit einem der Namen hinter except übereinstimmen. In unserem Beispiel nur einer, nämlich "ValueError:". Nachdem der Text der print-Anweisung ausgegeben wurde, führt die Ausführung eine weitere Schleife aus. Sie beginnt mit einer neuen Input().

Mehrere `except`-Klauseln

Eine `try`-Anweisung kann mehrere Handler für unterschiedliche Exception-Typen enthalten. Höchstens ein passender Handler wird ausgeführt.

Beim Öffnen und Auswerten einer Textdatei sind beispielsweise `OSError` (Datei-/Betriebssystemfehler) und `ValueError` (ungültige Zahl) naheliegende Fälle. `IOError` ist in modernem Python lediglich ein Alias von `OSError`.

```
try:
    with open("integers.txt", encoding="utf-8") as f:
        s = f.readline()
    i = int(s.strip())
except OSError as exc:
    print(f"I/O-Fehler: {exc}")
except ValueError:
    print("Keine gültige ganze Zahl.")
except Exception as exc:
    print("Unerwarteter Fehler:", type(exc).__name__, exc)
    raise
```

Die Variable `exc` ist an die konkrete Exception-Instanz gebunden. `OSError` besitzt bei Betriebssystemfehlern zusätzliche Attribute wie `errno`, `strerror` und gegebenenfalls `filename`. Häufig reicht es jedoch, die Exception direkt auszugeben.

Mehrere Exception-Typen können in einer `except`-Klausel als Tupel angegeben werden:

```
try:
    with open("integers.txt", encoding="utf-8") as f:
        i = int(f.readline().strip())
except (OSError, ValueError) as exc:
    print("Datei konnte nicht passend verarbeitet werden:", exc)
```

Wir wollen nun demonstrieren, was passiert, wenn wir eine Funktion innerhalb eines Try-Blocks aufrufen und innerhalb des Funktionsaufrufs eine Exception auftritt:

```
def f():
    x = int("four")
```

```
try:
    f()
except ValueError as e:
    print("Hab' den Fehler :-) ", e)


print("Weiter geht's")
```

```
Hab' den Fehler :-)  invalid literal for int() with base 10: 'four'
Weiter geht's
```

Hier wird die Ausnahme **nicht in `f()`**, sondern im aufrufenden `try`-Block abgefangen. Im nächsten Beispiel behandelt dagegen die Funktion selbst den `ValueError`:

```
def f():
    try:
        x = int("four")
    except ValueError as e:
        print("Fehler in Funktion :-) ", e)

try:
    f()
except ValueError as e:
    print("got it :-) ", e)


print("Weiter geht's")
```

```
Fehler in Funktion :-)  invalid literal for int() with base 10: 'four'
Weiter geht's
```

Im nächsten Schritt behandeln wir die Exception zunächst in der Funktion und lösen sie anschließend mit einem nackten `raise` **erneut aus**. Dadurch bleibt der ursprüngliche Traceback erhalten und die Ausnahme kann vom Aufrufer weiterbehandelt werden:

```
def f():
    try:
        x = int("four")
    except ValueError as e:
        print("Fehler in Funktion :-) ", e)
        raise

try:
    f()
except ValueError as e:
    print("got it :-) ", e)


print("Weiter geht's")
```

```
Fehler in Funktion :-)  invalid literal for int() with base 10: 'four'
got it :-)  invalid literal for int() with base 10: 'four'
Weiter geht's
```

Benutzerdefinierte Ausnahmen

Eigene Exception-Typen sind sinnvoll, wenn ein Anwendungsbereich einen eigenen Fehlerzustand ausdrücken soll. Benutzerdefinierte Ausnahmen sollten normalerweise von `Exception` erben, nicht direkt von `BaseException`. Ein sprechender Name mit dem Suffix `Error` erleichtert die Verwendung:

```
class UngueltigesAlterError(Exception):
    ⌋
    ↪  """Wird ausgelöst, wenn eine Altersangabe außerhalb des erlaubten Bereichs liegt."""


def pruefe_alter(alter):
    if not 0 <= alter <= 130:
        raise UngueltigesAlterError(f"Unplausibles Alter: {alter}")

pruefe_alter(150)
```

Der neue Exception-Typ kann anschließend genauso gezielt mit `except UngueltigesAlterError:` behandelt werden wie eine eingebaute Ausnahme.

```
try:
    pruefe_alter(150)
except UngueltigesAlterError as exc:
    print(exc)
```

Aufräum-Aktionen mit `finally`

Eine `finally`-Klausel wird beim Verlassen des zugehörigen `try`-Blocks ausgeführt – unabhängig davon, ob der Block normal endet, eine Ausnahme behandelt wird oder eine Ausnahme weiterpropagiert. `finally` **behandelt** eine Ausnahme jedoch nicht automatisch; ohne passenden `except`-Handler wird sie nach dem `finally`-Block weitergereicht.

Für Ressourcen wie Dateien ist häufig ein Context Manager (`with`) klarer als manuelles Aufräumen in `finally`.

```
try:
    x = float(input("Deine Zahl: "))
    inverse = 1.0 / x
finally:
    print("""Vielleicht ist eine Ausnahme
    aufgetreten oder auch nicht.""")
print("Die inverse Zahl: ", inverse)
```

```
Your number: 34
There may or may not have been an exception.
The inverse:  0.029411764705882353
```

Kombination von `try`, `except`, `else` und `finally`

- `except` behandelt passende Ausnahmen aus dem `try`-Block.

- `else` läuft nur, wenn im `try`-Block **keine** Ausnahme aufgetreten ist.
- `finally` läuft beim Verlassen des Konstrukts in jedem Fall.

Das `else` ist besonders nützlich, um Code, der selbst neue Ausnahmen auslösen kann, nicht unnötig in den geschützten `try`-Block aufzunehmen:

```
try:
    x = float(input("Deine Zahl: "))
    inverse = 1.0 / x
except ValueError:
    print("Das war keine Zahl!")
except ZeroDivisionError:
    print("Durch Null kann nicht dividiert werden.")
else:
    print("Die inverse Zahl:", inverse)
finally:
    print("Berechnung beendet.")
```

`else`-Klausel

Ein `else`-Block steht nach allen `except`-Klauseln und wird ausgeführt, wenn der `try`-Block ohne Ausnahme beendet wurde. Bei Dateioperationen lässt sich damit beispielsweise das Öffnen von der anschließenden Verarbeitung trennen:

```
file_name = "trullala.txt"
try:
    fh = open(file_name, encoding="utf-8")
except OSError as exc:
    print("Datei lässt sich nicht öffnen:", exc)
else:
    with fh:
        text = fh.readlines()
    print(f"{len(text)} Zeile(n) gelesen")
```

Noch kompakter ist es häufig, den Context Manager direkt innerhalb von `try` zu verwenden, wenn die gesamte Dateioperation als eine Einheit behandelt werden soll:

```
file_name = "fibonacci.py"
try:
    with open(file_name, encoding="utf-8") as fh:
        text = fh.readlines()
except OSError as exc:
    print("Datei lässt sich nicht öffnen:", exc)
else:
    print(f"{len(text)} Zeile(n) gelesen")
```

Die `assert`-Anweisung

`assert` ist für **interne Annahmen und Debugging-Invarianten** gedacht. Formal entspricht

```
assert bedingung, nachricht
```

bei aktivem Debug-Modus ungefähr:

```
if not bedingung:
    raise AssertionError(nachricht)
```

Assertions dürfen nicht für die Validierung externer Eingaben, Berechtigungsprüfungen oder andere zwingend notwendige Laufzeitkontrollen verwendet werden. Python kann Assertions bei optimierter Ausführung (`python -O`) vollständig entfernen.

```
x = 5
y = 3
assert x < y, "x has to be smaller than y"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-28-246a6fdef85a> in <module>
      1 x = 5
      2 y = 3
----> 3 assert x < y, "x has to be smaller than y"

AssertionError: x has to be smaller than y
```

`assert` ist also sinnvoll, um Annahmen des Programmierers über den internen Zustand zu dokumentieren und während Entwicklung und Tests zu prüfen. Erwartbare Benutzer- oder Dateifehler behandelt man dagegen mit normaler Validierung und geeigneten Exceptions.

```

```

Weitere Bücher von Bernd Klein

Die folgenden Verlagstitel sind eigenständige Veröffentlichungen beim Carl Hanser Verlag. Diese automatisch erzeugte PDF-Ausgabe von `python-kurs.eu` ist keine Hanser-Veröffentlichung.

Einführung in Python 3

Für Ein- und Umsteiger

Bernd Klein

ISBN: 978-3-446-46379-0

4. Auflage, 2021

Mehr beim Carl Hanser Verlag

Numerisches Python

Arbeiten mit NumPy, Matplotlib und Pandas

Bernd Klein

ISBN: 978-3-446-48584-6

3. Auflage, 2025

Mehr beim Carl Hanser Verlag

Funktionale Programmierung mit Python

Funktionale Konzepte praxisnah in Python einsetzen

Bernd Klein, Philip Klein

ISBN: 978-3-446-48191-6

1. Auflage, 2025

Mehr beim Carl Hanser Verlag

Hinweis zur Ausgabe

Diese Ausgabe wurde automatisiert aus den Quellen von `python-kurs.eu` erzeugt. Die Online-Fassung kann aktueller sein als diese PDF-Ausgabe.